AFRL-OSR-VA-TR-2014-0093

Stochastic Quantitative Reasoning for Autonomous Mission Planning

Carlos Varela
RENSSELAER POLYTECHNIC INST TROY NY

04/09/2014
Final Report

Air Force Research Laboratory
AF Office Of Scientific Research (AFOSR)/ RTC
Arlington, Virginia 22203
Air Force Materiel Command

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
31-03-2014	Final Performance Report	30-09-2011 TO 31-03-2014

4. TITLE AND SUBTITLE
Stochastic Quantitative Reasoning for Autonomous Mission Planning

5a. CONTRACT NUMBER

5b. GRANT NUMBER
FA9550-11-1-0332

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Varela, Carlos, A, Ph.D.

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Rensselaer Polytechnic Institute
110 8th Street
Troy, NY 12180

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
USAF, AFRL DUNS 143574726
AF OFFICE OF SCIENTIFIC RESEARCH
875 N. RANDOLPH ST. ROOM 3112
ARLINGTON VA 22203

10. SPONSOR/MONITOR'S ACRONYM(S)
AFOSR

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Distribution A - Approved for Public Release

13. SUPPLEMENTARY NOTES

14. ABSTRACT
In research performed with funding from this grant, PI Varela has developed mathematical concepts and software to automatically detect and correct for errors in spatio-temporal data streams. Varela and his group invented and formalized the notions of error signatures and mode likelihood vectors, and developed the PILOTS programming language v0.2.3. An important application of this work was to demonstrate that the Air France flight 447 accident from June 2009 could have been avoided by using these techniques applied on air speed, ground speed, and wind speed data streams.

15. SUBJECT TERMS
Spatio-temporal data streams, Error signatures, Mode likelihood vectors, PILOTS programming language, AF447 accident.

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			19b. TELEPHONE NUMBER *(Include area code)*

FINAL PERFORMANCE REPORT

Grant Title: Stochastic Quantitative Reasoning for Autonomous Mission Planning
Grant #: FA9550-11-1-0332
PI: Carlos A. Varela, Ph.D.
Program Manager: Frederica Darema, Ph.D.

Reporting Period: September 30, 2011 to March 31, 2014

Executive Summary:
In research performed with funding from this grant, PI Varela has developed mathematical concepts and software to automatically detect and correct for errors in spatio-temporal data streams. Varela and his group invented and formalized the notions of error signatures and mode likelihood vectors, and developed the PILOTS programming language v0.2.3. An important application of this work was to demonstrate that the Air France flight 447 (AF447) accident in June 2009 could have been avoided by using these techniques applied on air speed, ground speed, and wind speed data streams.

Archival Publications (published) during reporting period:

1. Carlos A. Varela. **Programming Distributed Computing Systems: A Foundational Approach**. MIT Press, May 2013.
2. Richard S. Klockowski, Shigeru Imai, Colin Rice, and Carlos A. Varela. **Autonomous Data Error Detection and Recovery in Streaming Applications**. In *Proceedings of the International Conference on Computational Science (ICCS 2013). Dynamic Data-Driven Application Systems (DDDAS 2013) Workshop*, pages 2036-2045, May 2013.
3. Matthew Newby, Nathan Cole, Heidi Jo Newberg, Travis Desell, Malik Magdon-Ismail, Boleslaw Szymanski, Carlos Varela, Benjamin Willett, and Brian Yanny. **A Spatial Characterization of the Sagittarius Dwarf Galaxy Tidal Tails**. *The Astronomical Journal*, 145(163), May 2013.
4. Shigeru Imai, Richard Klockowski, and Carlos A. Varela. **Self-Healing Spatio-Temporal Data Streams Using Error Signatures**. In *2nd International Conference on Big Data Science and Engineering (BDSE 2013)*, Sydney, Australia, December 2013.
5. Shigeru Imai, Thomas Chestna, and Carlos A. Varela. **Accurate Resource Prediction for Hybrid IaaS Clouds Using Workload-Tailored Elastic Compute Units**. In *6th IEEE/ACM International Conference on Utility and Cloud Computing (UCC 2013)*, Dresden, Germany, December 2013.
6. David Musser and Carlos A. Varela. **Structured Reasoning about Actor Systems**. In *Agere Workshop at ACM SPLASH 2013 Conference*, Indianapolis, Indiana, October 2013.
7. Carlos A. Varela, Manish Parashar, and Gul Agha, editors. **5th IEEE/ACM International Conference on Utility and Cloud Computing, UCC 2012, Chicago, IL, USA, November 5-8, 2012**, 2012. IEEE.

8. Shigeru Imai, Thomas Chestna, and Carlos A. Varela. **Elastic Scalable Cloud Computing Using Application-Level Migration**. In *5th IEEE/ACM International Conference on Utility and Cloud Computing (UCC 2012)*, Chicago, Illinois, USA, November 2012.
9. Shigeru Imai and Carlos A. Varela. **Programming Spatio-Temporal Data Streaming Applications with High-Level Specifications**. In *3rd ACM SIGSPATIAL International Workshop on Querying and Mining Uncertain Spatio-Temporal Data (QUeST) 2012*, Redondo Beach, California, USA, November 2012.
10. Shigeru Imai and Carlos A. Varela. **A Programming Model for Spatio-Temporal Data Streaming Applications**. In *Dynamic Data-Driven Application Systems (DDDAS 2012)*, ICCS, Omaha, Nebraska, pages 1139-1148, June 2012.
11. David Musser and Carlos A. Varela. **Human-Readable Machine-Checkable Abstract Reasoning about Actor Systems**. Technical report 12-01, Rensselaer Polytechnic Institute Department of Computer Science, January 2012.
12. Shigeru Imai. **Task Offloading between Smartphones and Distributed Computational Resources**. Master's thesis, Rensselaer Polytechnic Institute, May 2012.
13. Marco A. S. Netto, Christian Vecchiola, Michael Kirley, Carlos A. Varela, and Rajkumar Buyya. **Use of Run Time Predictions for Automatic Co-Allocation of Multi-Cluster Resources for Iterative Parallel Applications**. *Journal of Parallel and Distributed Computing*, pp 43pp, 2011.
14. Travis Desell, Malik Magdon-Ismail, Heidi Newberg, Lee A. Newberg, Boleslaw K. Szymanski, and Carlos A. Varela. **A Robust Asynchronous Newton Method for Massive Scale Computing Systems**. In *International Conference on Computational Intelligence and Software Engineering (CiSE)*, Wuhan, China, December 2011.
15. Shigeru Imai and Carlos A. Varela. **Light-Weight Adaptive Task Offloading from Smartphones to Nearby Computational Resources**. In *ACM Research in Applied Computation Symposium (RACS 2011)*, Miami, Florida, November 2011.
16. Qingling Wang and Carlos A. Varela. **Impact of Cloud Computing Virtualization Strategies on Workloads' Performance**. In *4th IEEE/ACM International Conference on Utility and Cloud Computing(UCC 2011)*, Melbourne, Australia, December 2011.
17. Shigeru Imai, Pratik Patel, and Carlos A. Varela. **Developing Elastic Software for the Cloud**. Invited book chapter in *Encyclopedia on Cloud Computing*, editors San Murugesan and Irena Bojanova, submitted, to appear mid-2014.
18. Shigeru Imai and Carlos A. Varela. **Dynamic Data-Driven Avionics Systems with Stochastic Error Detection and Correction**. Invited book chapter in *Dynamic Data Driven Application Systems (DDDAS)*, editor Frederica Darema, submitted, to appear mid-2014.

Students Involved in this Research:

1. Shigeru Imai, Ph.D. student
2. Richard Klockowski, Ph.D. student
3. Colin Rice, B.S. student
4. Alessandro Galli, B.S. student

Most Significant Advances and Conclusions:

The key research results directly supported by this grant are published in the following articles (available at http://wcl.cs.rpi.edu/bib/Keyword/DATA-STREAMING.html and also attached to this report):

1. Shigeru Imai, Richard Klockowski, and Carlos A. Varela. **Self-Healing Spatio-Temporal Data Streams Using Error Signatures**. In *2nd International Conference on Big Data Science and Engineering (BDSE 2013)*, Sydney, Australia, December 2013.
2. Richard S. Klockowski, Shigeru Imai, Colin Rice, and Carlos A. Varela. **Autonomous Data Error Detection and Recovery in Streaming Applications**. In *Proceedings of the International Conference on Computational Science (ICCS 2013). Dynamic Data-Driven Application Systems (DDDAS 2013) Workshop*, pages 2036-2045, May 2013.
3. Shigeru Imai and Carlos A. Varela. **Programming Spatio-Temporal Data Streaming Applications with High-Level Specifications**. In *3rd ACM SIGSPATIAL International Workshop on Querying and Mining Uncertain Spatio-Temporal Data (QUeST) 2012*, Redondo Beach, California, USA, November 2012.
4. Shigeru Imai and Carlos A. Varela. **A Programming Model for Spatio-Temporal Data Streaming Applications**. In *Dynamic Data-Driven Application Systems (DDDAS 2012)*, ICCS, Omaha, Nebraska, pages 1139-1148, June 2012.
5. Shigeru Imai and Carlos A. Varela. **Dynamic Data-Driven Avionics Systems with Stochastic Error Detection and Correction**. Invited book chapter in *Dynamic Data Driven Application Systems (DDDAS)*, editor Frederica Darema, submitted, to appear mid-2014.

Software and Data:

This report includes an attachment with all the PILOTS software and experimental data used during this research and reported in the published articles. The software is open-source and available at the following web site:

http://wcl.cs.rpi.edu/pilots/

Self-Healing Spatio-Temporal Data Streams Using Error Signatures

Shigeru Imai, Richard Klockowski, Carlos A. Varela
Department of Computer Science, Rensselaer Polytechnic Institute
110 Eighth Street, Troy, NY 12180, USA
Email: {imais,klockr,cvarela}@cs.rpi.edu

***Abstract*—Spatio-temporal data streams generated from sensors can be erroneous and could lead to serious problems. For example, pitot tubes icing which occurred to Air France flight 447 (AF447) in June 2009 led to faulty airspeed readings and eventually caused a fatal accident killing all 228 people on board. As an effort to develop self-healing spatio-temporal data stream systems, we have developed a highly declarative programming language called *PILOTS* that enables error detection and data correction based on *error signatures*. Error signatures are mathematical function patterns with constraints and are used to stochastically identify and categorize errors in redundant spatio-temporal data streams. In this paper, we refine the error detection and correction methods previously reported by the authors and apply these methods to real flight data of a private Cessna flight and the AF447 flight. The results show that the error detection and correction methods successfully work for both sets of flight data. For the private Cessna flight, three error scenarios are simulated: pitot tube failure, GPS failure, and simultaneous pitot tube and GPS failures. The error detection accuracy is approximately 93% and the response time to correct data is at most 5 seconds. For the AF447 flight, 162 seconds of available flight data including the pitot tubes failure is collected from the accident report and examined accordingly. The pitot tube failure of the AF447 flight is successfully detected and corrected after 5 seconds from the beginning of the failure. Overall error mode detection accuracy reaches 96.31%. Furthermore, our simulations show that the system never corrects data incorrectly, *i.e.*, all inaccurate mode detections produce either unknown or unrecoverable errors. These results suggest that the presented error signature-based detection and correction methods can fix erroneous data readings caused by sensor failures within a few seconds and thereby keep flight systems working properly. Such self-healing flight systems could have prevented the tragic AF447 accident from happening and saved the lives of all crew members and passengers.**

I. Introduction

Airplanes are one of the most complicated machines to operate since pilots have to deal with a lot of information provided from the instruments in a cockpit. In the event of instrument failures, making the right decision becomes even more difficult because of potentially partially erroneous data. In the worst case scenario, misinterpreting the data could lead to deadly accidents such as the Air France flight 447 (AF447) tragedy of 2009 in which 228 people were fatally injured [1].

The aircraft of the AF447 flight crashed in the Atlantic Ocean due to ice which temporarily formed in the pitot tubes causing erroneous airspeed readings, and the subsequent inability of the auto-pilot and human pilots to recover. The accident could have been prevented by endowing the flight system with the ability to understand the following data relationship:

$$\overrightarrow{v_g} = \overrightarrow{v_a} + \overrightarrow{v_w}. \quad (1)$$

where $\overrightarrow{v_g}$, $\overrightarrow{v_a}$, and $\overrightarrow{v_w}$ represent the *ground speed*, the *airspeed*, and the *wind speed* vectors. These speeds are obtained through *independent* data collection methods: the ground speed is typically computed from Global Positioning Satellite (GPS) system data, the airspeed is computed from air pressure measurements by pitot tubes, and the wind speed from weather forecast computer models. Since any one of the three speeds can be calculated using the other two with Equation (1), they are *redundant* to each other. Using the available redundancy in the data, we can detect and correct errors. Note that we use this speed example throughout the paper.

We have created a highly declarative programming language called *PILOTS* (**P**rogramm**I**ng **L**anguage for spati**O**-Temporal **S**treaming applications) [2], [3], [4] that enables data correction and detection of *spatio-temporal* data streams based on data redundancy. Spatio-temporal data streams refer to data streams whose items include associated spatial and temporal coordinates, often viewed as meta data. Examples include temperature measurements, financial stock values, gas prices, surveillance camera imaging, and aircraft sensor readings. A PILOTS program may specify 1) how to view heterogeneous data stream sources as homogeneous spatio-temporal data streams, 2) how to correct the data streams based on *error signatures*, and 3) how to output values of interest based on the corrected data streams. Error signatures are mathematical function patterns with constraints and are used to stochastically identify and categorize errors. The PILOTS programming language enables high-level development of applications to handle spatio-temporal data streams and ultimately assist humans in making better decisions.

The PILOTS project has evolved gradually to date. First, the design of the PILOTS programming language and the concept of error signatures were proposed [2]. Next, a runtime implementation of PILOTS capable of data selection and error signatures computation was presented [3]. Thirdly, an error detection method and a runtime implementation of PILOTS with error detection and correction capability were presented [4]. In this paper, we overview PILOTS version 0.2.3 [5] and mathematically refine the error signature-based detection and data correction methods. Also, we evaluate error

detection performance with real data of a private Cessna flight and the AF447 flight.

The rest of the paper is organized as follows. Section II describes technical background of the paper including methods and software for error detection and correction. Section III talks about error signatures for commonly used speed data in aviation and how to express these error signatures in PILOTS programs. Section IV shows performance metrics and results of error detection performance for a private Cessna flight and the AF447 flight data. Finally, we show related work in Section V and conclude the paper in Section VI with potential future directions.

II. Technical Background

A. Error Detection and Correction Methods

The error detection and correction methods [4] are refined and described in detail. The basic idea is that the algorithm recognizes the shape of an error function, identifies a type of error, and corrects associated data values if possible.

Error function An error function is an arbitrary function that computes a numerical value from independently measured input data. It is used to examine the validity of redundant data. If the value of an error function is zero, we interpret it as no error in the given data.

A vector $\overrightarrow{v}$ can be defined by a tuple (v, α), where v is the length of $\overrightarrow{v}$ and α is the angle between $\overrightarrow{v}$ and a base vector. Following this expression, $\overrightarrow{v_g}, \overrightarrow{v_a}$, and $\overrightarrow{v_w}$ are defined as $(v_g, \alpha_g), (v_a, \alpha_a)$, and (v_w, α_w) respectively as shown in Figure 1. To examine the relationship in Equation (1), we can compute $\overrightarrow{v_g}$ by applying trigonometry to $\triangle ABC$. We can define an error function as the difference between measured v_g and computed v_g as follows:

$$\begin{aligned} e(\overrightarrow{v_g}, \overrightarrow{v_a}, \overrightarrow{v_w}) &= |\overrightarrow{v_g} - (\overrightarrow{v_a} + \overrightarrow{v_w})| \\ &= v_g - \sqrt{v_a^2 + 2v_a v_w \cos(\alpha_a - \alpha_w) + v_w^2}. \end{aligned} \tag{2}$$

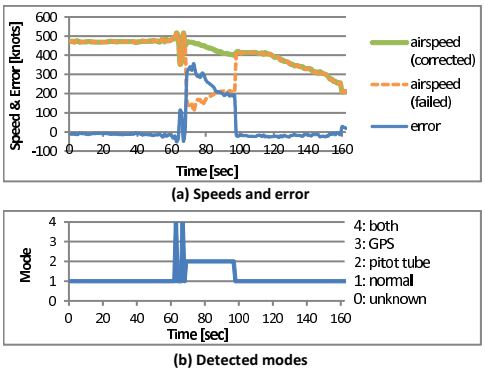


Fig. 1. Trigonometry applied to the ground speed, airspeed, and wind speed.

The values of input data are assumed to be sampled periodically from corresponding spatio-temporal data streams. Thus, an error function e changes its value as time proceeds and can also be represented as $e(t)$.

Error signatures An *error signature* is a constrained mathematical function pattern that is used to capture the characteristics of an error function $e(t)$ under a specific condition. Using a vector of constants $\bar{K} = \langle k_1, \ldots, k_m \rangle$, a function f, and a set of constraint predicates $\bar{P} = \{p_1(\bar{K}), \ldots, p_m(\bar{K})\}$, the error signature $S(\bar{K}, f(t), \bar{P}(\bar{K}))$ is defined as follows:

$$S(\bar{K}, f(t), \bar{P}(\bar{K})) = \{f(t) | p_1(\bar{K}) \wedge \cdots \wedge p_m(\bar{K})\}. \tag{3}$$

For example, an interval error signature can be defined as:

$$\begin{aligned} S_I(\bar{K}, f(t), \bar{I}(\bar{K}, \bar{A}, \bar{B})) \quad = \quad & \{f(t) | \\ & a_1 \leq k_1 \leq b_1, \ldots \\ & a_m \leq k_m \leq b_m\}, \end{aligned} \tag{4}$$

where $\bar{A} = \langle a_1, \ldots, a_m \rangle$ and $\bar{B} = \langle b_1, \ldots, b_m \rangle$. For example, when $f(t) = t + k, \bar{K} = \langle k \rangle, \bar{A} = \langle 2 \rangle$, and $\bar{B} = \langle 5 \rangle$, the error signature S_I contains all linear functions with slope 1, and crossing the Y-axis at values $[2, 5]$ as shown in Figure 2. On the other hand, for $f(t) = 0$, S_I only contains the constant function $f(t) = 0$.

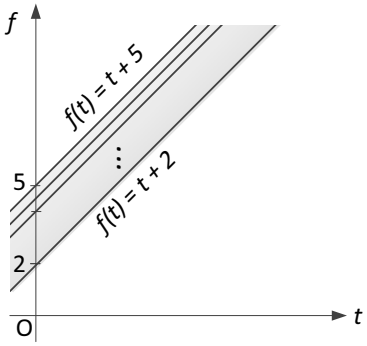


Fig. 2. Error signature S_I with a linear function $f(t) = t + k, 2 \leq k \leq 5$.

Given an error signature $S(\bar{K}, f(t), \bar{P}(\bar{K}))$, we enumerate its elements as *error signature samples*, *i.e.*,

$$s(t, \bar{K}) = f(t) \text{ s.t. } s(t, \bar{K}) \in S(\bar{K}, f(t), \bar{P}(\bar{K})). \tag{5}$$

An error signature sample is thus a particular function satisfying the constraints defined by an error signature. For the interval error signature S_I, a sample $s_I(t, \langle 3 \rangle)$ is $f(t) = t + 3$.

Mode likelihood vectors Given a set of error signatures $\{S_0, \ldots, S_n\}$, where S_0 corresponds to the normal mode signature with no errors, we calculate $\delta_i(t)$, the distance between the measured error function $e(t)$ and each error signature S_i by:

$$\delta_i(t) = \min_{\bar{K}} \int_{t-\omega}^{t} |e(t) - s_i(t, \bar{K})| dt. \tag{6}$$

where ω is the window size and $s_i(t, \bar{K}) \in S_i$. The smaller the distance $\delta_i(t)$, the closer the raw data is to the theoretical

signature S_i. We define the *mode likelihood vector* as $L(t) = \langle l_0(t), l_1(t), \ldots, l_n(t) \rangle$ where each $l_i(t)$ is defined as:

$$l_i(t) = \begin{cases} 1, & \text{if } \delta_i(t) = 0 \\ \frac{\min\{\delta_0(t), \ldots, \delta_n(t)\}}{\delta_i(t)}, & \text{otherwise.} \end{cases} \quad (7)$$

Observe that for each $l_i \in L, 0 < l_i \leq 1$ where l_i represents the ratio of the likelihood of signature S_i being matched with respect to the likelihood of the best signature. At each time stamp, the maximum two elements l_i and l_j of the mode likelihood vector, where $l_i \geq l_j$, are inspected in order to determine the error mode. Because of the way $L(t)$ is created, the maximum entry l_i will always be equal to 1. Given a threshold $\tau \in (0, 1)$ we check for one likely candidate that is sufficiently more likely than its successor by ensuring that $l_j \leq \tau$. Thus we determine the correct mode by choosing the error signature, and error mode i, corresponding to l_i which is S_i. If $i = 0$ then the system is in normal mode. If $l_j > \tau$, then regardless of the value of j, *unknown error* mode is assumed.

Error correction It is problem dependent if a known error mode i is recoverable or not. If there is a mathematical relationship between an erroneous value and other independently measured values, the erroneous value can be replaced by a new value computed from the other independently measured values. In the case of the speed example used in Equations (1) and (2), if the ground speed v_g is detected as erroneous, its corrected value v_g^c can be computed by the airspeed and wind speed as follows:

$$v_g^c = \sqrt{v_a^2 + 2 v_a v_w \cos(\alpha_a - \alpha_w) + v_w^2}. \quad (8)$$

B. Error Detection and Correction Software

PILOTS (**P**rogramm**I**ng **L**anguage for spati**O**-**T**emporal data **S**treaming applications) is a programming language specifically designed for analyzing data streams incorporating space and time. Using PILOTS, application developers can easily program an application that handles spatio-temporal data streams by writing a high-level (declarative) program specification. The PILOTS code includes an *inputs* section to specify the data streams and how data is to be extrapolated from incomplete data, typically using declarative geometric criteria (*e.g.*, *closest*, *interpolate*, *euclidean* keywords) [3]. It includes *outputs* and *errors* sections to specify the data streams to be produced by the application, as a function of the input streams with a given frequency. If a detected error is recoverable, output values are computed from corrected input data, otherwise original input data is used. The *signatures* and *correct* sections, enable PILOTS programmers to specify error signatures for known error conditions, as well as the function to use to correct the data automatically if such data errors are found.[1]

Figure 3 shows the architecture of the PILOTS runtime system, which implements the error detection and correction methods described in the previous section. It consists of three parts: the *Data Selection*, the *Error Analyzer*, and the *Application Model* modules.

The Application Model obtains homogeneous data streams $(d'_1, d'_2, \ldots, d'_N)$ from the Data Selection module, and then it generates outputs $(o_1, o_2, \ldots, o_M)$ and data errors $(e_1, e_2, \ldots, e_L)$. The Data Selection module takes heterogeneous incoming data streams $(d_1, d_2, \ldots, d_N)$ as inputs. Since this runtime is assumed to be working on moving objects, the Data Selection module is aware of the current location and time. Thus, it returns appropriate values to the Application Model by selecting or interpolating data in time and location depending on the data selection method specified in the PILOTS program.

The ErrorAnalyzer collects the latest ω error values from the Application Model and keeps analyzing errors based on the error signatures. If it detects a recoverable error, then it replaces an erroneous input with the corrected one by applying a corresponding error correction equation. The Application Model computes the outputs based on the corrected inputs produced from the Error Analyzer.

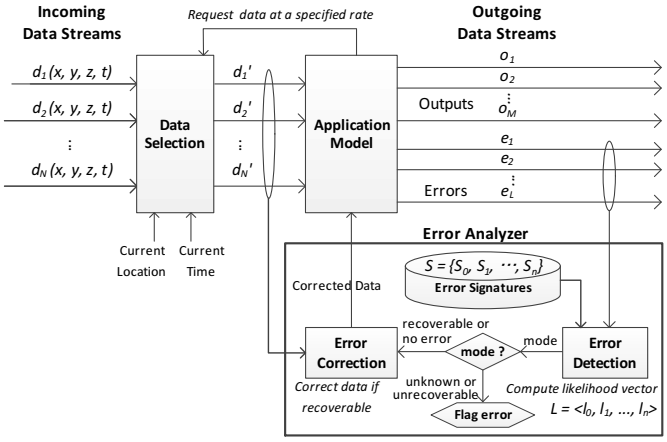


Fig. 3. Data streaming architecture with error detection and correction.

III. Error Signatures for Self-Healing Speed Data

In this section, we derive a set of error signatures for the speed example used in the previous sections. Also, we present a PILOTS program implementing the error signatures and corresponding error correction equations.

A. Error Signatures

We consider the following four error modes: 1) normal (no error), 2) pitot tube failure due to icing, 3) GPS failure, 4) both pitot tube and GPS failures. Suppose the airplane is flying at airspeed v_a. For computing error signatures for different error conditions, we will assume that other speeds as well as failed airspeed and ground speed can be expressed as follows.

- ground speed: $v_g \approx v_a$.
- wind speed: $v_w \leq a v_a$, where a is the wind to airspeed ratio.

[1]Parameters τ and ω—for specifying threshold and time window respectively—can be given in command-line options.

- pitot tube failed airspeed: $b_l v_a \leq v_a^f \leq b_h v_a$, where b_l and b_h are the lower and higher values of pitot tube clearance ratio and $0 \leq b_l \leq b_h \leq 1$. 0 represents a fully clogged pitot tube, while 1 represents a fully clear pitot tube.
- GPS failed ground speed: $v_g^f = 0$.

We assume that when a pitot tube icing occurs, it is gradually clogged and thus the airspeed data reported from the pitot tube also gradually drops and eventually remains at a constant speed while iced. This resulting constant speed is characterized by ratio b_l and b_h. On the other hand, when a GPS failure occurs, the ground speed suddenly drops to zero. This is why we model the failed ground speed as $v_g^f = 0$.

In the case of pitot tube failure, let the ground speed, wind speed, and airspeed be $v_g = v_a, v_w = av_a$, and $v_a^f = bv_a$. The error function (2) can be expressed as follows:

$$e = v_a - \sqrt{v_a^2(b^2 + 2ab\cos(\alpha_a - \alpha_w) + a^2)}.$$

Since $-1 \leq \cos(\alpha_a - \alpha_w) \leq 1$, the error is bounded by the following:

$$\begin{aligned} v_a - \sqrt{v_a^2(a+b)^2} \leq \quad & e \quad \leq v_a - \sqrt{v_a^2(a-b)^2} \\ (1-a-b)v_a \leq \quad & e \quad \leq (1-|a-b|)v_a. \end{aligned} \tag{9}$$

In the case of GPS failure, let the ground speed, wind speed, and airspeed be $v_g^f = 0, v_w = av_a$, and $v_a = v_a$. The error function (2) can be expressed as follows:

$$e = 0 - \sqrt{v_a^2(1 + 2a\cos(\alpha_a - \alpha_w) + a^2)}.$$

Similarly to the pitot tube failure, we can derive the following error bounds:

$$-(a+1)v_a \leq \quad e \quad \leq -|a-1|v_a. \tag{10}$$

We can derive error bounds for the normal and both failure cases similarly. Applying the wind to airspeed ratio a and the pitot tube clearance ratio $b_l \leq b \leq b_h$ to the constraints obtained in Inequations (9) and (10), we get the error signatures for each error mode as shown in Table I.

TABLE I
ERROR SIGNATURES FOR SPEED DATA.

Mode	Error Signature	
	Function	Constraints
Normal	$e = k$	$k \in [-av_a, av_a]$
Pitot tube failure	$e = k$	$k \in [(1-a-b_h)v_a, (1-\lvert a-b_l\rvert)v_a]$
GPS failure	$e = k$	$k \in [-(a+1)v_a, -\lvert a-1\rvert v_a]$
Both failures	$e = k$	$k \in [-(a+b_h)v_a, -\lvert a-b_l\rvert v_a]$

When $a = 0.1, b_l = 0.2$, and $b_h = 0.33$, the error signatures shown in Table I are visually depicted in Figure 4.

B. PILOTS program

A PILOTS program called `speedcheck` implementing the error signatures shown in Table I is presented in Figure 5. This program checks if the wind speed, airspeed, and ground speed are correct or not, and computes a crab angle, which is used to adjust the direction of the aircraft to keep a desired ground track. For this program to be applicable to a Cessna 182-RG, we use a cruise speed of 162 knots as v_a. Each section of the program is explained in order:

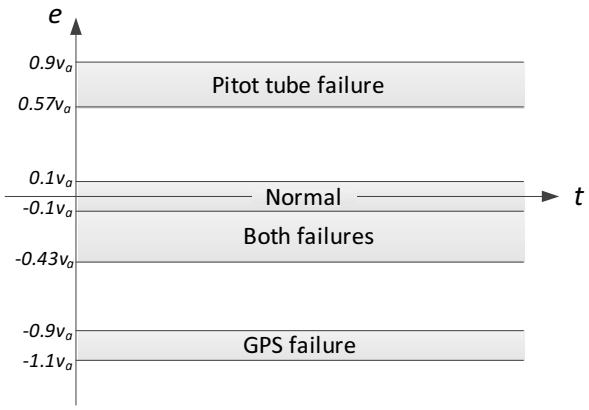


Fig. 4. Error Signatures for speed data ($a = 0.1, b_l = 0.2$, and $b_h = 0.33$).

- *inputs:* All the speed and angle data required to compute the error and crab angle are defined here with data selection methods. Since heterogeneous input data streams of `air_speed, air_angle, ground_speed` and `ground_angle` are defined for 2D regions and time, `euclidean(x,y)` and `closest(t)` select data which is closest to the current location in 2D euclidean space and then closest to the current time. For `wind_speed` and `wind_angle`, since they are defined for 3D regions and time, `interpolate(z,2)` is finally used to get linearly interpolated values in the Z-axis using two data points after `euclidean(x,y)` and `closest(t)` are applied.
- *outputs:* The crab angle and corrected speed data are computed every second.
- *errors:* The error function `e` defined in Equation (2) is computed. The angle signs are reversed in the formulae, because in mathematics, angles increase counterclockwise (with 0° representing East) while in aviation, angles increase clockwise (with 0° representing North).
- *signatures:* There are four error signatures {`S0, S1, S2, S3`} associated with the error function `e`. They are all constrained by a constant `k` with lower and upper bounds based on the error signatures shown in Table I.
- *correct:* The error modes 1 and 2, which are identified by `S1` and `S2`, can be corrected using the equations defined for the airspeed and ground speed. If the error mode 3 corresponding to `S3` is detected, it is not possible to correct two variables at the same time, thus this error is unrecoverable.

IV. EVALUATION

We apply the error signatures defined in Section III to two sets of real flight data. The first one is a private flight using

```
program speedcheck;
  inputs
    wind_speed, wind_angle (x,y,z,t) using
      euclidean(x,y), closest(t), interpolate(z,2);
    air_speed, air_angle (x,y,t) using
      euclidean(x,y), closest(t);
    ground_speed, ground_angle (x,y,t) using
      euclidean(x,y), closest(t);

  outputs
    crab_angle:
      arcsin(wind_speed * sin(wind_angle - air_angle) /
        sqrt(air_speed^2 + 2 * air_speed * wind_speed *
          cos(wind_angle - air_angle) + wind_speed^2))
            at every 1 sec;
    air_speed_out:    air_speed at every 1 sec;
    ground_speed_out: ground_speed at every 1 sec;
    wind_speed_out:   wind_speed at every 1 sec;

  errors
    e: ground_speed -
         sqrt(air_speed^2 + wind_speed^2 + 2 * air_speed *
           wind_speed * cos(wind_angle - air_angle));

  signatures
    /* v_a = 162 knots */
    S0(k): e=k,  -16.2<=k, k<=  16.2 "Normal";
    S1(k): e=k,   91.8<=k, k<= 145.8 "Pitot tube failure";
    S2(k): e=k, -178.2<=k, k<=-145.8 "GPS failure";
    S3(k): e=k,  -70.2<=k, k<= -16.2 "Both failures";

  correct
    S1: air_speed = sqrt(ground_speed^2 + wind_speed^2
          2 * ground_speed * wind_speed *
            cos(ground_angle - wind_angle));
    S2: ground_speed = sqrt(air_speed^2 + wind_speed^2
          2 * air_speed * wind_speed *
            cos(wind_angle - air_angle));
end
```

Fig. 5. A declarative specification of the `speedcheck` PILOTS program.

a Cessna 182-RG identified by N756VH [6] from Albany, NY to Fort Meade, MD on April 3rd, 2012. The other is the Air France flight 447 using an Airbus A330-203 which took off from Rio de Janeiro bound for Paris on June 1st, 2009. To simulate the failures mentioned in Section III, we added corresponding errors to the N756VH Cessna flight data; however, we used the real pitot tube failure data for the AF447 flight. PILOTS programs' error detection accuracy and response time to mode changes are evaluated.

A. Performance Metrics

- **Accuracy**: This metric is used to evaluate how accurately the algorithm determines the true mode. Assuming the true mode transition $m(t)$ is known for $t = 0, 1, 2, \ldots, T$, let $m'(t)$ for $t = 0, 1, 2, \ldots, T$ be the mode determined by the error detection algorithm. We define $accuracy(m, m') = \frac{1}{T}\sum_{t=0}^{T} p(t)$, where $p(t) = 1$ if $m(t) = m'(t)$ and $p(t) = 0$ otherwise.
- **Maximum/Minimum/Average Response Time**: This metric is used to evaluate how quickly the algorithm reacts to mode changes. Let a tuple (t_i, m_i) represent a mode change point, where the mode changes to m_i at time t_i. Let $M = \{(t_1, m_1), (t_2, m_2), \ldots, (t_N, m_N)\}$ and $M' = \{(t'_1, m'_1), (t'_2, m'_2), \ldots, (t'_{N'}, m'_{N'})\}$ be the sets of true mode changes and detected mode changes respectively. For each $i = 1 \ldots N$, we can find the smallest t'_j such that $(t_i \leq t'_j) \wedge (m_i = m'_j)$; if not found, let t'_j be t_{i+1}. The response time r_i for the true mode m_i is given by $t'_j - t_i$. We define the maximum, minimum, and average response times by $\max_{1 \leq i \leq N} r_i$, $\min_{1 \leq i \leq N} r_i$, and $\frac{1}{N}\sum_{i=1}^{N} r_i$ respectively.

B. Experiment 1: N756VH Cessna Flight

1) Flight data: Flight data is collected through the following independent sources:

- **ground speed**: Flight track log provided by FlightAware [6].
- **airspeed**: Manually recorded by the pilot.
- **wind speed**: Weather forecast information provided by National Weather Service [7].

The flight duration is 1 hour 41 minutes. The collected speed data and error computed by Equation (2) are shown in Figure 6. Notice that the airspeed data during take off and landing is not accurate due to the data collection mechanism.

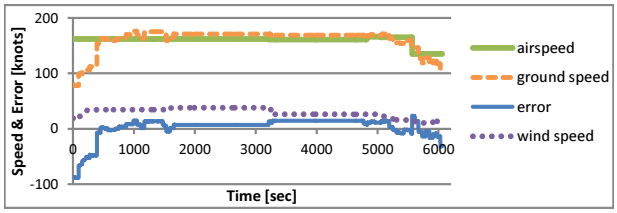


Fig. 6. Collected speeds and error for the N756VH 03-Apr-2012 KALB-KFME flight (normal).

2) Experimental Settings: Using the `speedcheck` PILOTS program shown in Figure 5, the 6060 seconds (=1 hour 41 minutes) of flight departing from Albany, NY and landing at Fort Meade, MD are recreated. Three types of error are simulated as shown below. In each case, all data streams except for erroneous one(s) are actual. Defined error modes are: 0 for unknown, 1 for normal, 2 for pitot tube failure, 3 for GPS failure, and 4 for both failures.

- **Pitot tube failure**: 2400 seconds after the departure, the airspeed drops from 162 knots to 50 knots within 10 seconds and stays at 50 knots until landing. The set of true mode changes is given by $M = \{(1, 1), (2401, 2)\}$.
- **GPS failure**: 2400 seconds after the departure, the ground speed drops from 171 knots to 0 knots immediately and stays at 0 knots until landing. The set of true mode changes is given by $M = \{(1, 1), (2401, 3)\}$.
- **Both pitot tube and GPS failures**: The above two speed changes happen simultaneously at 2400 seconds after the departure. Both speeds remain failed until landing. The set of true mode changes is given by $M = \{(1, 1), (2401, 4)\}$.

To find out the effect of the window size ω and threshold value τ on the accuracy and response time, we measure these metrics for window sizes $\omega \in \{1, 2, 4, 8, 16\}$ and threshold $\tau \in \{0.2, 0.4, 0.6, 0.8\}$. Note that since there is only one error mode change in each true mode changes set, we can get only one response time result for each simulated error case.

3) Results: Results of the accuracy and response time are are shown in Figure 7. For all the three cases, when $\omega = 1$ and $\tau = 0.8$, the best results are observed as follows: accuracy = 0.9294 and response time = 4 seconds for the pitot tube failure, accuracy = 0.935 and response time = 0 seconds for the GPS failure, and accuracy = 0.9342 and response time = 5 seconds for both failures. Accuracy is not even higher due to airspeed data during takeoff and landing which was not collected because the pilot was busy operating the airplane, which makes the system incorrectly detect a both failure mode. Because the airspeed gradually drops, it takes a few seconds to detect it as a pitot tube failure; however, a GPS failure is immediately detected since the ground speed promptly drops to zero when it happens. This is why the response time for the GPS failure is better than the other two cases. Since the used error signature sets are non-overlapping constant functions (*i.e.*, $e = k$), even though smaller window sizes are normally noise-prone compared to bigger window sizes, past data is not necessary to determine the correct error modes. In this experiment, noise on the error is not big enough to jump out of the boundaries defined by error signature sets, therefore $\omega = 1$ gives the best results.

In Figure 7(b-1) for the GPS failure, when $\tau = 0.2$, accuracy is unusually low compared to the other two failure cases. This occurs because too low a threshold makes the normal and GPS failure modes compete against each other in the landing phase and thus the resulting mode falls into unknown mode for the last 600 seconds.

The transitions of the corrected speed and detected modes that show the best accuracy are shown in Figures 8 (pitot tube failure), 9 (GPS failure), and 10 (both failures) respectively. For the first 390 seconds, the error mode is detected wrongly in all three cases; the true modes are 1 (normal mode) whereas the detected modes are 4 (both failures) during this period. These wrong mode detections are originated from the erroneously recorded airspeed. Other than that, the error detection method works pretty well for all three cases.

Detected modes go into the unknown mode for a short period around 2401 seconds for both pitot tube failure and both failures. Since the airspeed takes a few seconds to drop, during that time, the normal and pitot tube failure modes are competing against each other for the pitot tube failure case. For the both failures case, the GPS failure and both failures modes are competing. Unlike the other two cases, the ground speed drops immediately for the GPS failure, and there is no conflict with other error modes, thus the GPS failure mode is correctly detected without going into the unknown mode.

C. Experiment 2: Air France Flight 447

1) Flight Data: The ground speed and airspeed are collected based on Appendix 3 in the final report of Air France flight 447 [1]. Note that the (true) airspeed was not recorded in the flight data recorder so that we computed it from recorded Mach (M) and static air temperature (SAT) data. The airspeed was obtained by using the relationship: $v_a = a_0 M \sqrt{SAT/T_0}$, where a_0 is the speed of sound at standard sea level (661.47

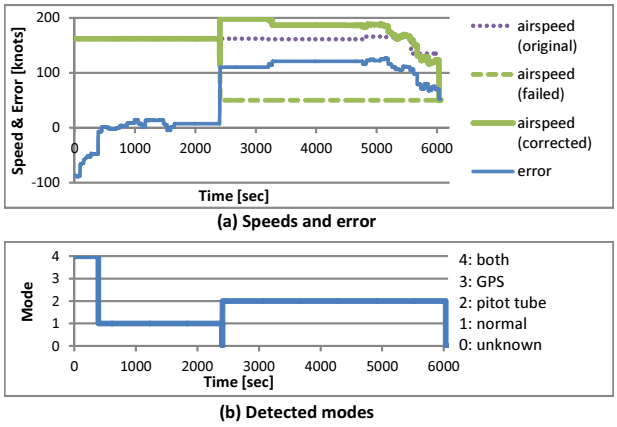


Fig. 8. Corrected airspeed and detected modes for the N756VH 03-Apr-2012 KALB-KFME flight (pitot tube failure, $\tau = 0.8, \omega = 1$).

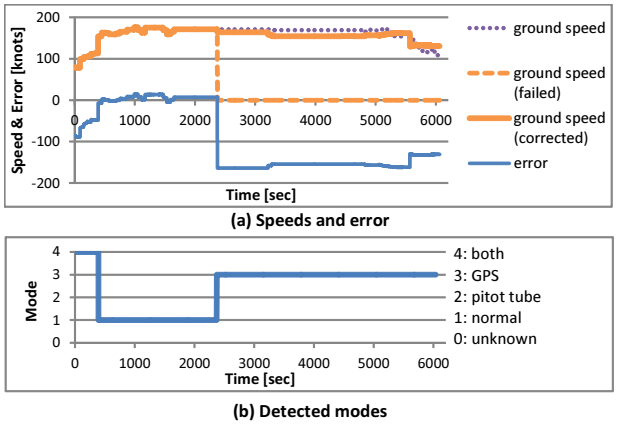


Fig. 9. Corrected ground speed and detected modes for the N756VH 03-Apr-2012 KALB-KFME flight (GPS failure, $\tau = 0.8, \omega = 1$).

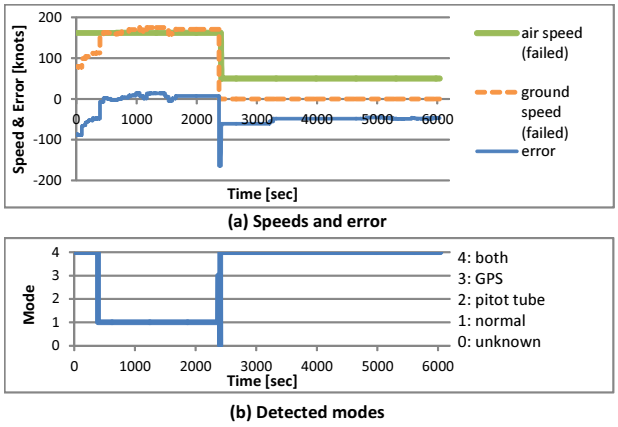


Fig. 10. Uncorrected speeds and detected modes for the N756VH 03-Apr-2012 KALB-KFME flight (pitot tube and GPS failure, $\tau = 0.8, \omega = 1$).

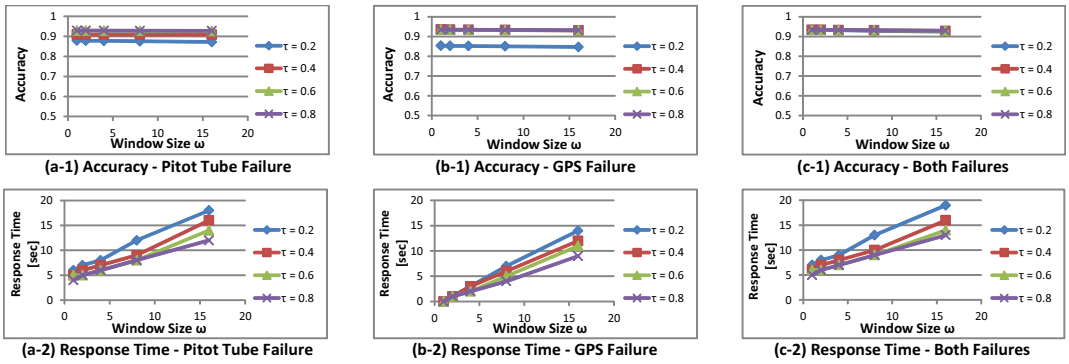


Fig. 7. Accuracy and response time for the N756VH 03-Apr-2012 KALB-KFME flight

knots) and T_0 is the temperature at standard sea level (288.15 Kelvin). Independent wind speed information was not recorded either. According to the description from page 47 of the final report: "(From the weather forecast) the wind and temperature charts show that the average effective wind along the route can be estimated at approximately *ten knots tail-wind*." We followed this description and created the wind speed data stream as ten knots tail wind.

2) Experimental Settings: According to the final report, speed data was provided from 2:09:00 UTC on June 1st 2009 and it became invalid after 2:11:42 UTC on the same day. Thus, we examine the valid 162 seconds of speed data including a period of pitot tube failure which occurred from 2:10:03 to 2:10:36 UTC. We also use the `speedcheck` PILOTS program shown in Figure 5 except for constraints values in *signatures* which use $v_a = 470$ knots, the cruise airspeed of the AF447 flight. Defined error modes are the same as Experiment 1, so the set of true mode changes is defined as $M = \{(1,1),(64,2),(98,1)\}$. The accuracy and average response time are investigated for window sizes $\omega \in \{1,2,4,8,16\}$ and threshold $\tau \in \{0.2,0.4,0.6,0.8\}$.

3) Results: Results of the accuracy and maximum/minimum/average response times are shown in Figure 11. Same as Experiment 1, the best results, accuracy = 0.9631, maximum/minimum/average response times = 5/0/2.5 seconds, are observed when $\omega = 1$ and $\tau = 0.8$. Overall trends of the accuracy and response time are same as Experiment 1 because of the nature of the error signature set.

The transitions of the corrected speed and detected modes that show the best accuracy with $\omega = 1$ and $\tau = 0.8$ are shown in Figure 12. Looking at Figure 12(b), the pitot tube failure is successfully detected from 69 to 97 seconds except for the interval 64 to 69 seconds due to the slowly decreasing airspeed. The response time for the normal to pitot tube failure mode is 5 seconds and for the pitot tube failure to normal mode is 0 seconds (thus the average response time is 2.5 seconds). From Figure 12(a), the airspeed successfully starts to get corrected at 69 seconds and seamlessly transitions to the normal airspeed when it recovers at 98 seconds.

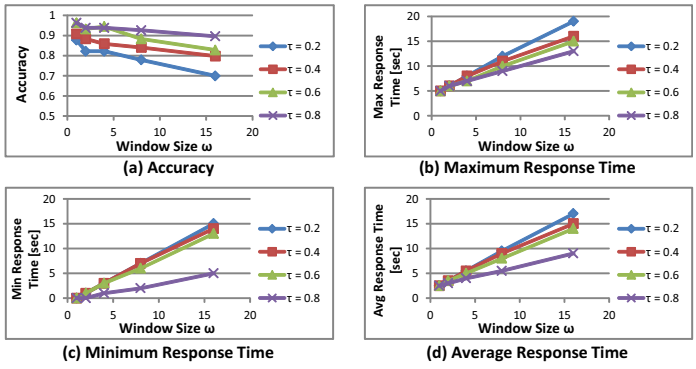


Fig. 11. Accuracy and response time for AF447 flight.

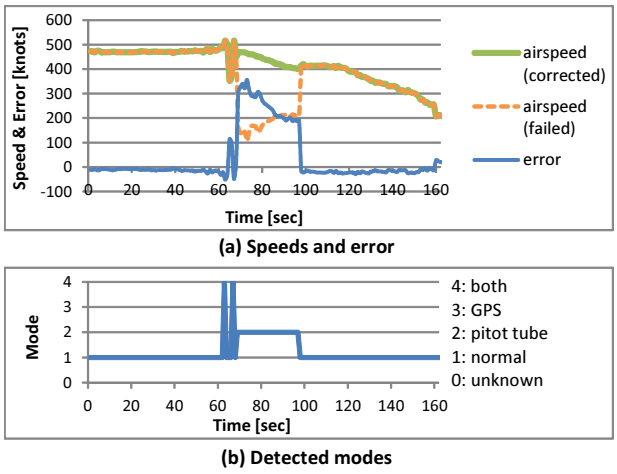


Fig. 12. Corrected airspeed and detected modes for AF447 flight.

V. Related Work

There are several systems that combine stream processing and data base management, *i.e.*, Data Stream Management Systems or DSMS, such as STREAM [8], Aurora [9], and TelegraphCQ [10]. They are designed to execute SQL-like queries to unbounded continuous incoming data streams and output events of interest. Microsoft StreamInsight is a DSMS-

based system and has been extended to support spatio-temporal streams [11]. Also, the concept of the moving object data base (MODB) which adds support for spatio-temporal data streaming to DSMS is discussed in [12]. These DSMS-based spatio-temporal stream management systems support general continuous queries for multiple moving objects. Our streaming data analytics to detect errors based on signatures and correct data on the fly is beyond the scope of a purely declarative SQL-based query approach. Furthermore, our domain-specific approach enables highly declarative description of input-output relationships between streams, error functions, error signatures, and data correction functions using the PILOTS programming language.

Distributed streaming systems have been studied in the context of cloud computing [13], [14]. Our data error correction methods could be useful for distributed settings as well by connecting multiple distributed PILOTS applications.

VI. Conclusion and Future Directions

We define a general error signature set for aviation speed data and evaluate error detection performance of PILOTS programs with real flight data. We find that the accuracy and response time improve as the threshold τ increases. The reason of this behavior is that there are some cases in which there are two competing modes whose likelihood values are close to each other, and due to the closeness, the mode detection algorithm tends to regard it as an unknown error mode. Higher threshold values are more tolerant to multiple competing modes, thus give better results. Unsurprisingly, there is a positive correlation between the window size and response times for all the threshold values. This is an intuitive result because the less the error detection algorithm uses past data, the more responsive it becomes to mode changes. In addition, a faster average response time leads to a better accuracy result since the error detection algorithm cannot predict mode changes, but only react to them. That is, a smaller window size implies better accuracy. This is true because our designed error signature set produces nearly orthogonal mode likelihood vectors. Also, it is noteworthy that our error detection and data correction methods never correct data incorrectly.

When computing mode likelihood vectors, time to compute distances by Equation (6) can be significant due to the exponential growth of the search space as the size of the constants set $\bar{K}$ increases. To use the presented error detection and correction methods in larger-scale real-time systems, techniques to bound the running time must be devised.

Future research directions include applying the error signature-based error detection and correction methods to other flight accidents, *e.g.*, those fuel sensor reading errors. Also developing PILOTS flight systems that process real-time data from external sources such as 3D terrain data, updated weather, and information from other airplanes. More and more data are expected to be available in cockpits in the near future [15], and thus automated data analysis systems will become even more crucial to both manned and unmanned aerial vehicles. We envision smarter and safer flight systems processing massive data in real-time. Such systems need to reason about spatial and temporal data and constraints and give the pilots better information to make more accurate judgments during critical moments. The presented techniques and software can be used as a promising starting point to develop these flight systems.

Acknowledgments

This research is partially supported by Air Force Office of Scientific Research Grant No. FA9550-11-1-0332 and a Yamada Corporation Fellowship.

References

[1] Bureau d'Enquêtes et d'Analyses pour la Sécurité de l'Aviation Civile, "Final Report: On the accident on 1st June 2009 to the Airbus A330-203 registered F-GZCP operated by Air France flight AF 447 Rio de Janeiro - Paris," http://www.bea.aero/en/enquetes/flight.af.447/rapport.final.en.php, 2012.

[2] S. Imai and C. A. Varela, "A programming model for spatio-temporal data streaming applications," in *Dynamic Data-Driven Application Systems (DDDAS 2012)*, Omaha, Nebraska, June 2012, pp. 1139–1148.

[3] ——, "Programming spatio-temporal data streaming applications with high-level specifications," in *3rd ACM SIGSPATIAL International Workshop on Querying and Mining Uncertain Spatio-Temporal Data (QUeST) 2012*, Redondo Beach, California, USA, November 2012.

[4] R. S. Klockowski, S. Imai, C. Rice, and C. A. Varela, "Autonomous data error detection and recovery in streaming applications," in *Dynamic Data-Driven Application Systems (DDDAS 2013) Workshop*, May 2013, pp. 2036–2045.

[5] Worldwide Computing Laboratory, Rensselaer Polytechnic Institute, "The PILOTS programming language," http://wcl.cs.rpi.edu/pilots/.

[6] FlightAware, "Flight track log for N756VH on 03-Apr-2012 (KALB-KFME)," http://flightaware.com/live/flight/N756VH/history/20120403/1800Z/KALB/KFME/tracklog.

[7] NOAA's National Weather Service, "Forecast winds and temps aloft," http://aviationweather.gov/products/nws/winds/.

[8] A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, K. Ito, R. Motwani, U. Srivastava, and J. Widom, "Stream: The Stanford data stream management system," in *ACM SIGMOD Conference*. Springer, 2004.

[9] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik, "Aurora: a new model and architecture for data stream management," *The VLDB JournalThe International Journal on Very Large Data Bases*, vol. 12, no. 2, pp. 120–139, 2003.

[10] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, F. Reiss, and M. A. Shah, "TelegraphCQ: continuous dataflow processing," in *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*. ACM, 2003, pp. 668–668.

[11] M. H. Ali, B. Chandramouli, B. S. Raman, and E. Katibah, "Spatio-temporal stream processing in Microsoft StreamInsight," *IEEE Data Eng. Bull.*, pp. 69–74, 2010.

[12] K. An and J. Kim, "Moving objects management system supporting location data stream," in *Proceedings of the 4th WSEAS International Conference on Computational Intelligence, Man-Machine Systems and Cybernetics*, ser. CIMMACS'05. Stevens Point, Wisconsin, USA: World Scientific and Engineering Academy and Society (WSEAS), 2005, pp. 99–104.

[13] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari, "S4: Distributed stream computing platform," in *Data Mining Workshops (ICDMW), 2010 IEEE International Conference on*. IEEE, 2010, pp. 170–177.

[14] V. Gulisano, R. Jimnez-Peris, M. Patio-Martnez, C. Soriente, and P. Valduriez, "StreamCloud: An elastic and scalable data streaming system," *IEEE Trans. Parallel Distrib. Syst.*, pp. 2351–2365, 2012.

[15] U.S. Department of Transportation Federal Aviation Administration, "Code of federal regulations part 91.225: Automatic dependent surveillance-broadcast (ADS-B) out performance requirements to support air traffic control (ATC) service; final rule," http://www.faa.gov/regulations_policies/faa_regulations/, July 2013.

Available online at www.sciencedirect.com
ScienceDirect
Procedia Computer Science 00 (2013) 1–10
Procedia Computer Science

International Conference on Computational Science, ICCS 2013

Autonomous Data Error Detection and Recovery in Streaming Applications

Richard Klockowski, Shigeru Imai, Colin L Rice, Carlos A. Varela*

Computer Science Department, Rensselaer Polytechnic Institute, 110 Eighth Street, Troy, NY 12180, USA

Abstract

Detecting and recovering from errors in data streams is paramount to developing successful autonomous real-time streaming applications. In this paper, we devise a multi-modal data error detection and recovery architecture to enable automated recovery from data errors in streaming applications based on available redundancy. We formally define *error signatures* as a way to identify classes of abnormal conditions and *mode likelihood vectors* as a quantitative discriminator of data stream condition modes. Finally, we design an extension to our own declarative programming language, PILOTS, to include error correction code. We define performance metrics for our approach, and evaluate the impact of monitored data window size and mode likelihood change threshold on the accuracy and responsiveness of our data-driven multi-modal error detection and correction software. Tragic accidents—such as Air France's flight from Rio de Janeiro to Paris in June 2009 killing all people on board— can be prevented by implementing auto-pilot systems with an airspeed data stream error detection and correction algorithm following the fundamental principles illustrated in this work.

Keywords: redundant data error correction, spatio-temporal data streams, programming languages

1. Introduction

We present a software framework for developing resilient data driven applications and systems that act upon redundant spatio-temporal data streams. In this work we assume a spatio-temporal data streaming application model, where input streams associated to space and time get converted into output streams and error streams according to a mathematical description of the behavior of the application. Much like redundant bits in error correcting hardware, stream redundancies allow for dynamic detection and correction of *known* types of failures. Redundancy is a key aspect present in many spatio-temporal data streaming applications. However, unless it is effectively used by systems, autonomous recovery from error conditions is not possible. There are many complex ways in which a set of redundant input streams may fail. We propose a system towards automatically correcting known failures that can be detected in the source streams. We formalize *error signatures*, mathematical function patterns that enable autonomous systems to accurately detect when an erroneous condition exists in an input data stream. A multi-modal architecture uses these error signatures to switch each stream between different modes of operation. *Mode likelihood vectors* are computed

*Corresponding author
Email address: cvarela@cs.rpi.edu (Carlos A. Varela*)

in real-time by interpolating streamed data to a set of known error signatures. These vectors are used to determine the condition that input streams are exhibiting. When in a known error condition mode, the erroneous original data stream is automatically replaced by a data stream that is computed from the redundant (correct) data streams. The system dynamically adapts to errors in the data streams by switching modes, and it can resume normal behavior when input data is no longer categorized as being erroneous. We design an extension to PILOTS, a declarative programming language to not only compute error signatures from high-level specifications of spatio-temporal data streaming applications, but also to enable these applications to recover from known errors by using available redundancy in the data.

The Air France AF447 accident in June 2009 left 12 crew members and 216 passengers dead [1]. The reason for the crash was faulty sensor data that caused the automatic pilot to disengage, ultimately confusing the human pilots who were unable to take timely corrective actions. The pitot tubes of the airplane began to freeze which caused incorrect air speed readings, switching the plane from *normal law* to *alternate law*, and eventually causing the pilots to enter an unintended fatal stall. After a technical investigation by the Bureau d'Enquêtes et d'Analyses pour la Sécurité de l'Aviation Civile (BEA) it is clear that this error condition is detectable and can be corrected by an active redundant data-driven flight system. We argue that disasters like this are preventable by using an automatic pilot that implements the framework described in this paper: a multi-modal dynamic data-driven error correction software framework using error signatures and mode likelihood vectors.

2. Data Error Detection and Correction Architecture

Our contribution is an autonomous error correcting architecture for data streaming applications (depicted in Figure 1). The architecture was designed for applications with redundant input streams. For a set of input streams $D = \{d_1, d_2, ..., d_n\}$, the redundancy of the streams can be defined as the set of functions $R = \{r_1, r_2, ..., r_m\}$ where each r_i is a function $r_i(\hat{d}_1, \ldots, \hat{d}_k) = d_j$ for $j \in [1...n], k < n$, $\hat{d}_1, \ldots, \hat{d}_k \in D$ and $d_j \notin \{\hat{d}_1, \ldots, \hat{d}_k\}$. An *error function* associated to a particular input stream d_j may take the form $e_j = d_j - r_i(\hat{d}_1, \ldots, \hat{d}_k)$, where r_i is the redundancy function $r_i(\hat{d}_1, \ldots, \hat{d}_k) = d_j$. We define *error signatures* as the shape of these error functions for *previously known* error conditions. Additionally we formalize the *mode likelihood vector*, which is used to determine whether there is an error and whether or not it can be corrected. Data stream error correction is provided in the case that a redundancy within the other working streams is available. Especially in the case of spatio-temporal streams that use inherently redundant physical data such as those found in a flight system using sensor data, error signatures enable developing effective real-time error warning and correction systems. We contend that our proposed software framework can be useful to prevent tragedies such as the Air France plane crash. For this purpose we are developing PILOTS: a programming language for spatio-temporal data streaming applications [2]. PILOTS allows us to view heterogeneous data streams as homogeneous by declaratively selecting data according to geometric principles. In this paper, we describe an extension to the language design to include error correction code using the notion of error signatures. This software should prove very helpful for streaming application developers to enable them to create effective error correcting software.

2.1. Error Signatures

The purpose of an error signature is to be able to reason about which data stream may contain an error. A collection of error signatures, called an *error signature set*, is matched against the observed error which provides a means of error detection. We assume the existence of an *error function* which is simply a function of the input streams that captures the redundancy in the data streams. The *measured* error for an application is the value of the error function over a window of time. Each error signature corresponds to a particular type of failure in the input streams. The effectiveness of error signatures is highly dependent on the choice of error function. When there are no problems with the input streams, error functions typically evaluate to zero.

An error signature describes the behavior of the error function under particular operating conditions which we choose to call *modes*. An important distinction is made between *theoretical* error signatures, which correspond to known error modes, and *measured* error which is generated by looking at the raw input data. Theoretical error signatures are currently defined as a function of time which may contain constants $k_0, \ldots, k_n$ satisfying a set of constraints. In order to identify useful error signatures for a particular application, we currently employ an empirical method of simply running a simulation using data that exhibits a certain type of error and observe the results in the *measured*

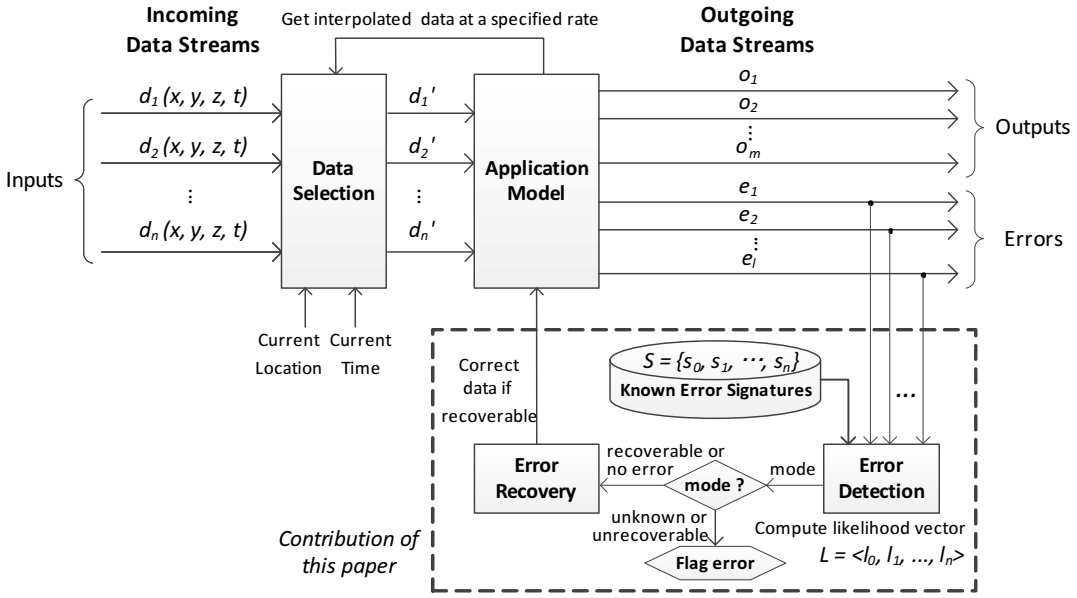


Figure 1: Data streaming architecture with error detection and correction

error. The error signature under *normal* conditions signifies that no errors have been detected. When all input streams are working properly the system assumes *normal* mode. Otherwise one of three modes is assumed: *unknown*, *recoverable*, or *unrecoverable*. If the system reaches *recoverable* mode, an error signature has been matched with the observed error and the appropriate redundancy is available to replace the stream producing the error. Thus for each error signature there exists a corresponding mode. If no redundancy is available the system switches to *unrecoverable* mode where a flag is raised (e.g., a red light bulb) denoting the type of error that was detected. In *unknown* error mode a similar type of flag is raised, but there is no known error signature that corresponds to the observed error. Only specific types of errors, those which have distinct error signatures and place the system into a recoverable mode, can be detected and corrected.

2.2. Error Detection

The measured error is compared to each of the theoretical error signatures in an attempt to find a strong match. Our current method for comparing error signatures is accomplished by formulating what we call the *mode likelihood vector*. Let $\{s_0, \ldots, s_n\}$ be the collection of known theoretical error signatures, where s_0 corresponds to the normal mode signature with no errors. We calculate the distance vector $\Delta(t) =< \delta_0(t), \ldots, \delta_n(t) >$ where $\delta_i(t)$ is the distance between the measured error $e(t)$ and $s_i(t)$. Specifically, $\delta_i(t) = \int_{t-\omega}^{t} |e(t) - s_i(t)|dt$ where $e(t)$ is the measured error and ω is the window size. The smaller the distance, the closer the raw data is to the theoretical signature. We formally define the mode likelihood vector to be $L(t) =< l_0(t), l_1(t), \ldots, l_n(t) >$ where each $l_i(t)$ is defined as:

$$l_i(t) = \begin{cases} 1, & \text{if } \delta_i(t) = 0 \\ \frac{\min\{\delta_0(t),\ldots,\delta_n(t)\}}{\delta_i(t)}, & \text{otherwise.} \end{cases}$$

Observe that for each $l_i \in L$ it follows that $0 \leqslant l_i \leqslant 1$, where l_i represents the ratio of the likelihood of signature s_i being matched with respect to the likelihood of the *best* signature. At each time stamp, the maximum two elements l_j and l_k of the mode likelihood vector, where $l_j > l_k$, are inspected in order to determine the error mode. Because of the way $L(t)$ is created, the maximum entry l_j will always be equal to 1. Given a threshold $\tau \in (0, 1)$ we check for one likely candidate that is sufficiently more likely than its successor by ensuring that $l_k \leqslant \tau$. Consequently, a *known* error mode is assumed. The correct mode is determined by choosing the error signature, and error mode, corresponding to l_j which is s_j. Each recoverable error mode uniquely determines the input streams that are erroneous. If $j = 0$ then

the system is in normal mode. If $l_k > \tau$ then, regardless of the value of j, *unknown error* mode is assumed and an error flag is raised. No corrective action can be taken because the measured error cannot be recognized, and the input data flows through the application uncorrected. A well-behaved set of error signatures will produce *nearly orthogonal* mode likelihood vectors, where one element is a one and the rest are close to zero. In sections 3 and 4 we study the impact of the choice of theoretical error signature sets on detection and correction results.

2.3. Error Recovery

If we assume that the system is in one of the known error modes (i.e., a match has been found for the measured error) then an attempt can be made at correcting the error. Recall that the *error function* is given and contains information about the redundancy between data streams. If an input stream d_j experiences an error and a redundancy r_i exists which can replace that stream, then the error is recoverable. After the error has been corrected, the original input streams will continue to be monitored to determine if the error has subsided and the system is able to reenter normal mode.

3. Twice: A Case Study

We explore how error signatures affect the values of mode likelihood vectors defined in Section 2 by using a very simple data streaming application called *Twice*.

3.1. A Simple Data Streaming Application

Twice is a simple data streaming application which takes two input data streams, a and b, where b is supposed to be twice as large as a, and outputs an error defined by $b-2*a$. Stream data for a and b are expected to increase by one for a and by two for b every second (i.e., $a(t) = t$ and $b(t) = 2 * t$), so the error is zero in the *normal* case; however, several modes of errors could happen depending on different types of failures as shown in Figure 2. Figure 2(a) shows *normal* mode, where most of the time the error remains zero, but there are several spikes due to transient fluctuation of the data input timing. Figure 2(b) suggests critical failure of a's data source. We will call this *a failure* mode. At around 50 seconds of the simulation time, the error starts growing linearly. This linear increase of the error explains that a remains a constant value whereas b continues increasing its value. Similarly, Figure 2(c) shows a situation where a correctly increases, but b fails to increase its value. Similarly, we will call this *b failure* mode. Figure 2(d) shows an example of an *out-of-sync* mode, where the error becomes consistently large at around 30 seconds of the simulation time. This is because a's input data stream becomes consistently one second behind b's input data stream.

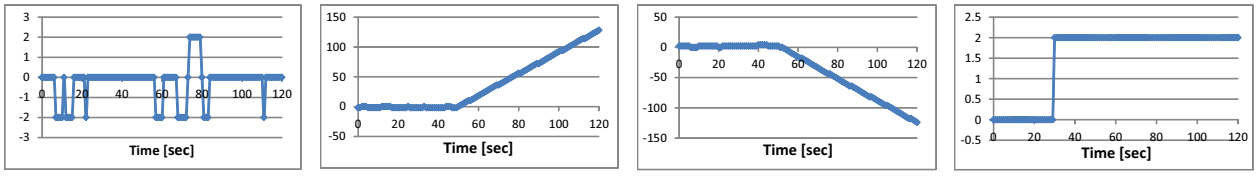


Figure 2: Known error patterns for twice example

3.2. Error Signatures for Twice

To correctly differentiate the four different modes of errors presented in Figure 2, we define error signatures for each mode. In the course of our case study, we evaluated three sets of error signatures as shown in Table 1. For the *no error*, *a failure*, and *b failure* modes, all error signatures are the same in these error signature sets. That is, $e = 0$ for no error, $e = 2t + k$ for *a failure*, and $e = -2t + k$ for *b failure*. Each error signature is designed to capture a characteristic pattern of error we see in the previous section. For example, an error signature for *a failure* is a linear function with a slope of 2 and a constant k, which resembles the increasing line starting at around 50 seconds of *a failure* shown in Figure 2(b). Differences among error signature sets are limited to the *out-of-sync failure* mode. Both *base* and *out-of-sync restricted* error signature sets have $e = k$ for the *out-of-sync mode*, but the *out-of-sync restricted* imposes a constraint on the value of k ($|k| > \tau_{oos}$). The τ_{oos} threshold is intended to prevent noise and small out-of-sync conditions from being categorized as abnormal.

Table 1: Error signature sets defined for *twice* example

Error signature set	Mode			
	No error	A failure	B failure	Out-of-sync
Base	$e = 0$	$e = 2t + k$	$e = -2t + k$	$e = k$, where $k \neq 0$
Out-of-sync restricted				$e = k$, where $\|k\| > \tau_{oos}$
Out-of-sync removed				none

3.3. Mode Estimation Study

Using the error signatures defined in Table 1, we estimate the operating modes for a 480 seconds sequence of measured error including mode change within sixty-second intervals as shown in Figure 3(a). Figure 4(a) shows the ground truth: the transition of modes that is used to generate the streams. In Figure 4, modes are mapped to 0 for *unknown*, 1 for *no error*, 2 for *a failure*, 3 for *b failure*, and 4 for *out-of-sync*. For each set of error signatures presented in the previous section, we first compute the likelihood of each mode, and then estimate mode likelihoods relative to the maximum likelihood which represents the minimum signature interpolation distance.

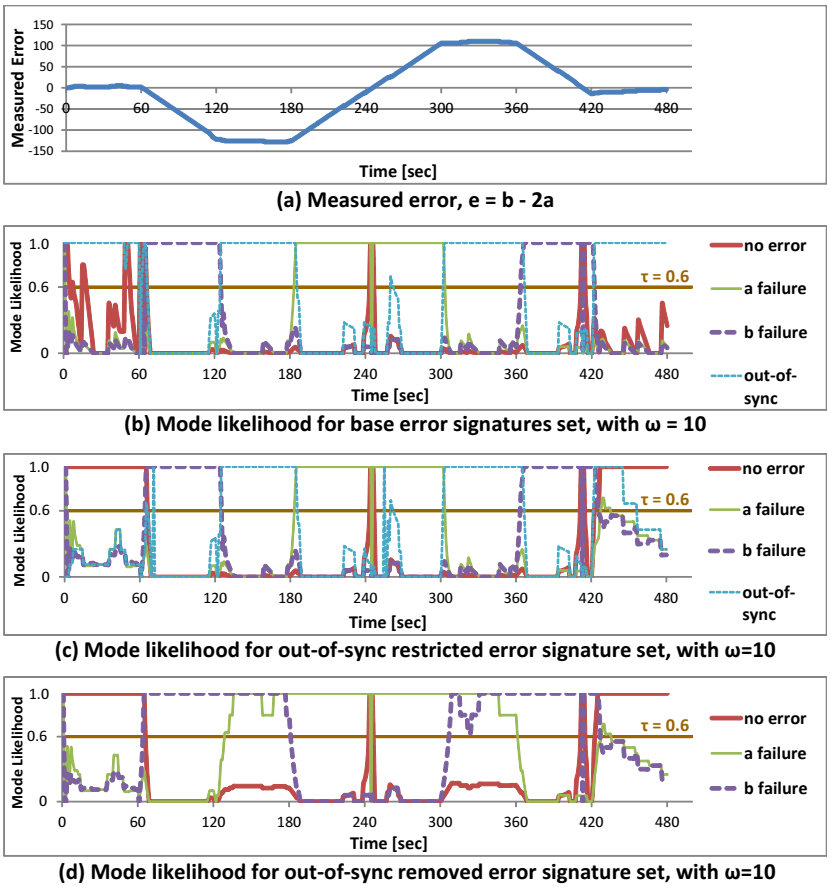


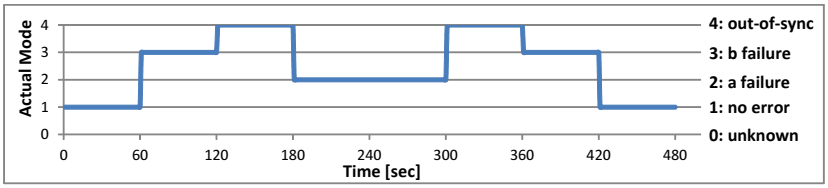
Figure 3: Measured error and mode likelihood results for twice example

The results of the mode likelihood and the estimated modes are shown in Figure 3(b)-(d) and 4(b)-(d) respectively. Figure 3(b) and 4(b) are results for the base error signatures, Figure 3(c) and 4(c) are results for the *out-of-sync restricted* error signatures, and Figure 3(d) and 4(d) are results for the *out-of-sync removed* error signatures.

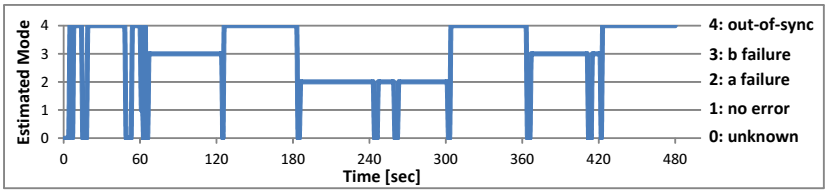
- **Base:** Looking at the result of estimated mode in Figure 4(b), most of the first and last 60 seconds are recognized as the *out-of-sync mode*, where they are actually supposed to be in the *normal* mode. This incorrect estimation

occurs because the given error in Figure 3(a) contains noise so that the actual value of the error is not always exactly zero. Thus, the *out-of-sync* error signature fits better to the measured error than *no error* since the constant k in the *out-of-sync* error signature can be any value other than zero. This essentially illustrates an ill-defined error signature set: since *normal* and *out-of-sync* conditions are difficult to distinguish from each other, computed mode likelihood vectors are far from orthogonal.

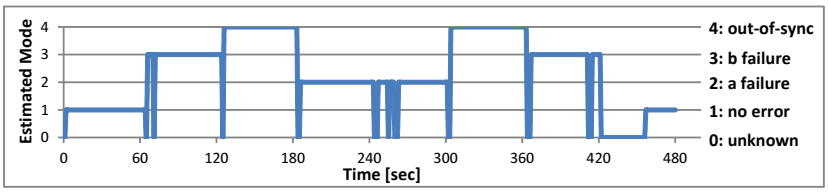
- **Out-of-sync restricted:** The result of estimated mode in Figure 4(c) looks closer to the true mode in Figure 4(a) than the result of the *base* error signature set. The threshold τ_{oos} ($\tau_{oos} = 20$ for this experiment) is a constraint on the *out-of-sync* error signature that prevents it from matching the first and last 60 seconds, and correctly lets the *normal* signature match those periods instead.
- **Out-of-sync removed:** The result of estimated mode in Figure 4(d) does not match the *out-of-sync* mode at around 120-180 and 300-360 seconds since that mode does not exist, but it successfully goes into *unknown* error mode which are valid estimations when using this signatures set. Also, it succeeds to match *normal* mode at around 0-60 and 420-480 seconds most of the time.



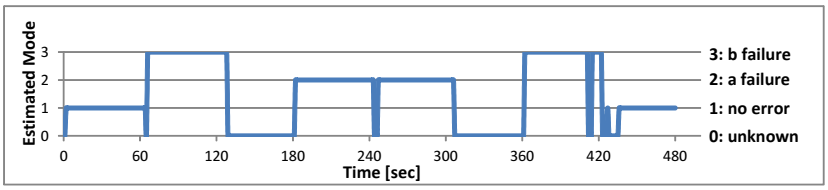
(a) Ground truth for generated stream data



(b) Estimated mode for base error signatures set, with τ = 0.6



(c) Estimated mode for out-of-sync restricted error signatures set, with τ = 0.6



(d) Estimated mode for out-of-sync removed error signatures set, with τ = 0.6

Figure 4: Estimated modes for twice example

4. Performance Metrics and Experimental Results

We evaluate the performance of the proposed error detection algorithm which depends on the window size, ω, representing how much historical data we consider, and the minimum likelihood threshold τ, representing how well data must match a single signature in order to select a corresponding operation mode.

4.1. Performance Metrics

- **Accuracy**: This metric is used to evaluate how accurately the algorithm determines the true mode. Assuming the true mode transition $m(t)$ is known for $t = 0, 1, 2, ..., T$, let $m'(t)$ for $t = 0, 1, 2, ..., T$ be the mode determined by the error detection algorithm. We define $accuracy(m, m') = \frac{1}{T}\sum_{t=0}^{T} p(t)$, where $p(t) = 1$ if $m(t) = m'(t)$ and $p(t) = 0$ otherwise.
- **Average Response Time**: This metric is used to evaluate how quickly the algorithm reacts to mode changes. Let a tuple (t_i, m_i) represent a mode change point, where the mode changes to m_i at time t_i. Let $M = \{(t_1, m_1), (t_2, m_2), ..., (t_{N_1}, m_{N_1})\}$ and $M' = \{(t'_1, m'_1), (t'_2, m'_2), ..., (t'_{N_2}, m'_{N_2})\}$ be the sets of true mode changes and detected mode changes respectively. We compute the average response time as shown in Algorithm 1.

input : True mode changes $M = \{(t_1, m_1), (t_2, m_2), ..., (t_{N_1}, m_{N_1})\}$, t_{N_1+1} = simulation end time,
Detected mode changes $M' = \{(t'_1, m'_1), (t'_2, m'_2), ..., (t'_{N_2}, m'_{N_2})\}$
output: Average response time $AvgResp$

$Responses = 0$;
for $i \leftarrow 1$ **to** N_1 **do**
  Find the smallest t'_j such that $(t_i \leqslant t'_j) \wedge (m_i = m'_j)$; if not found, $t'_j \leftarrow t_{i+1}$;
  $Responses = Responses + (t'_j - t_i)$;
end
return $AvgResp = Responses/N_1$;

Algorithm 1: Average response time computation

4.2. Experimental Results

Based on the metrics defined in the previous section, we evaluate the monitored error for the twice example in Figure 3(a) with five different random seeds for noise generation. We use each of the error signature sets for evaluation: *base*, *out-of-sync restricted*, and *out-of-sync removed.* For the *base* and *out-of-sync restricted* error signature sets, a set of true mode changes is given by $M = \{(1, 1), (61, 3), (121, 4), (181, 2), (301, 4), (361, 3), (421, 1)\}$. However, for the *out-of-sync removed* error signature, we replace the *out-of-sync* errors with *unknown* errors (respectively 4 and 0 on the y-axis of Figure 4) in M. We do this for fairness because the *out-of-sync removed* error signatures cannot detect an *out-of-sync* error at all. To find out the effect on the accuracy and average response time by the window size ω and threshold value τ, we measure these metrics for window size $\omega \in \{5, 10, 15, 20\}$ and threshold $\tau \in \{0.2, 0.4, 0.6, 0.8\}$.

Accuracy and average response time results for the *base*, *out-of-sync restricted*, and *out-of-sync removed* error signature sets are shown in Figure 5. For all the three error signature sets, there is a trend that the accuracy and average response time improve as the threshold τ increases. This result can be explained by the following: there are some cases in which there are two competing modes whose likelihood values are close to each other, and due to the closeness, the mode detection algorithm tends to regard it as an unknown error mode. Higher threshold values are more permissive, thus give better results in this example. However if the choice of τ is too large then this system may choose to enter a known error mode when the correct choice is actually unknown error mode.

There is a positive correlation between the window size and average response time for all the threshold values. This is an intuitive result: the less the algorithm uses past data, the more responsive it becomes to mode changes. Also, a faster average response time leads to a better accuracy result since the error detection algorithm cannot predict mode changes, but only react to them. That is, a smaller window size implies better accuracy. In fact, the *base* and *out-of-sync restricted* error signature sets take the best accuracy/average response time when the window size is the smallest $\omega = 5$ and threshold $\tau = 0.8$. On the other hand, in the case of the *out-of-sync removed* error signature set, the accuracy and average response time are peaked when window size $\omega = 10$. Thus, the most appropriate window size is different depending on each error signature set.

The *out-of-sync removed* error signature set works best. By analyzing three different sets of error signatures for this simple example, we see the importance of the error signature set to get accurate mode estimation results quickly. Especially, as we can see in the results from the *base* error signatures set, error signatures should not be very close in terms of error patterns they match, otherwise those error signatures are vulnerable to noise. *Well-behaved*

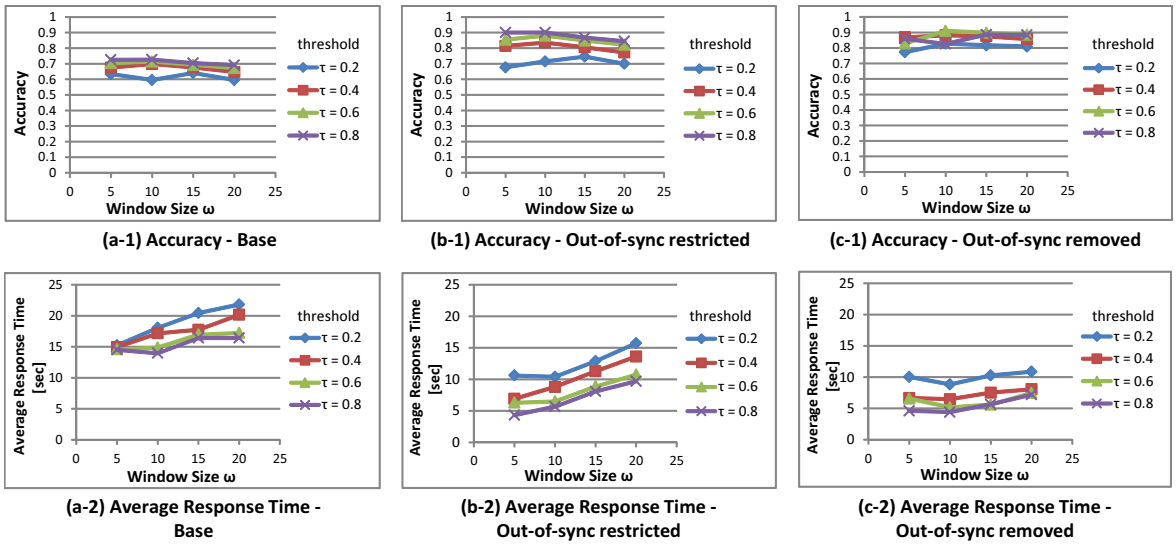


Figure 5: Results of accuracy and average response time for *base*, *out-of-sync restricted*, and *out-of-sync removed* error signature sets

error signatures are those that, under *normal* and *known* error conditions, produce near orthogonal mode likelihood vectors.

5. Implementation with A Spatio-Temporal Data Streaming Programming Language

PILOTS (ProgrammIng Language for spatiO-Temporal data Streaming applications) is a programming language specifically designed for analyzing data streams incorporating space and time, as in applications running on moving objects [3, 2]. Using PILOTS, application developers can easily program an application that handles spatio-temporal data streams by writing a high-level declarative program specification. The system architecture for applications implemented in the PILOTS programming language is shown in Figure 1: everything outside of the dotted box. In this architecture, the application gets data $(d'_1, d'_2, \ldots, d'_n)$ from the *data selection* module. This takes incoming heterogeneous spatio-temporal data streams $(d_1, d_2, \ldots, d_n)$ and outputs homogeneous data streams depending on the current location and time, and the application generates output $(o_1, o_2, \ldots, o_m)$ and data errors $(e_1, e_2, \ldots, e_l)$ based on an *application model*. Whereas spatio-temporal data is available with various spatial density and time frequency depending on data sources, applications often need to process data at a constant frequency. To view such heterogeneous data streams as *homogeneous* data streams, the data selection module specifically provides first-class support for data selection and interpolation so that applications can get data consistently regardless of the data's original spatio-temporal heterogeneity.

We extend the PILOTS programming language to incorporate an error correction method. Two new keywords, `signatures` and `correct`, are introduced in addition to the existing PILOTS grammar defined in [3] to specify which data streams have an associated redundancy and how to correct the incoming data. The statements under the `signature` keyword describe the application's error signature set. Each statement has a label containing any constant parameters, a functional description of the error signature, and an optional list of constraints on the constant parameters separated by commas. The statements under the `correct` keyword declare the relationship between a particular error signature, the corresponding erroneous stream, and the redundancy available to fix the error. This information is enough to know how to handle recoverable error modes. If a data error is detected when matching a known error signature, we can correct an erroneous input as specified under `correct`. If a signature is not included in the `correct` clause, then it is a *known* but *unrecoverable* error. Here we explain how error corrections can be written in the program specification by using the *Twice* example, as shown in Figure 6. The error correction support for PILOTS is realized by the *error detection* module, depicted in Figure 1, which takes all the error outputs $(e_1, e_2, ..., e_l)$ and tries to detect erroneous data inputs by comparing the error outputs with the *known error signatures*. If an error

on data input d_i' is detected, it will be replaced by the value specified in the `correct` clause by the *error recovery* module.

```
program twice;
    inputs
        a: (t) using closest(t);
        b: (t) using closest(t);
    outputs;
    errors
        e: (b - 2 * a) at every 1 sec;
    signatures
        s0:    e = 0;
        s1(k): e = 2*t + k;
        s2(k): e = -2*t + k;
        s3(k): e = k, abs(k) > 20;
    correct
        s1(k): a = b / 2;
        s2(k): b = 2 * a;
end;
```

Figure 6: A simple program specification with error correction

6. Related Work

First and foremost this work builds upon the programming language PILOTS [2, 3] which targets spatio-temporal data streaming applications such as those found in flight systems. The detailed investigation reported in [1] suggests that our notion of error signatures to detect data errors can be quite useful. The concept of the moving object data base (MODB) which supports spatio-temporal data streaming is discussed in [4]. This research is relevant because many applications of error signatures will include data corresponding to a moving object (such as a plane).

Stream processing has become very attractive in the last decade. Surveys on general data streaming applications and methods include [5, 6]. The concept of the rule engine is discussed in [5] which has many similarities to our error signatures system but does not correct the input streams. General-purpose data stream management systems [7, 8, 9] cannot afford the declarative specification of data streams and data error correction that our domain-specific approach provides. In [10] a component is added to stream processing systems to *orchestrate* the behavior of the applications, including correcting domain specific errors. However this is an event driven system, the input data is not being directly monitored. To incorporate error correction using the notion of error signatures into a distributed environment, the work presented in [11] for setting up a set of distributed stream processing systems may be useful. A distributed data processing framework can help with the performance and scalability of data analyses.

7. Conclusion

In this paper we devised a multi-modal data error detection and recovery architecture based on our definition of *error signatures* as mathematical function patterns. We defined *mode likelihood vectors* as a quantifiable measure of the likelihood of the application being in a normal or a particular error mode, as defined by an error signature. Well-behaved error signatures are those that produce orthogonal mode likelihood vectors on *normal* and *known* error conditions. Ill-defined error signatures (those producing non-orthogonal vectors) lead to more undesirable or incorrect *unknown error* mode conditions, rendering our error detection and correction framework less useful. Real-time analysis of error streams and pattern matching against known error signatures enables streaming applications to switch from normal operation mode into *known error modes*. If the known error is *recoverable*, thanks to the redundancy available in the data, we autonomously correct the faulty data stream, so that applications continue to behave normally. Furthermore, we continue monitoring the input streams, so that normal operation can be reinstated when data are considered no longer erroneous.

Accuracy and responsiveness depend on the *window size*, ω, of the monitored data, and on the *threshold*, τ, imposed on the relative likelihood of a mode before accepting a change in the application's mode of operation. Using a

simple streaming application we found that, the larger ω is, the less responsive (higher response time) the algorithm. However if ω is too small, the system enters *unknown* mode more frequently affecting both accuracy and responsiveness. When the signature set is well-behaved, τ has less effect on accuracy, since mode likelihood vectors will be near orthogonal. However, for less well-behaved signature sets, smaller values of τ will cause the system to enter *unknown* error mode more often, while larger values of τ will produce more false positives. Since, the requirements on accuracy and responsiveness are ultimately application-dependent, application developers need to find the right balance of these parameters to tune their applications' error detection and correction behavior. The implementation of the extended PILOTS programming language, due to its declarative nature, will help quickly prototype new applications and develop better error detection and correction methodologies.

Future work includes creating well-behaved error signatures for aeronautical applications in order to correct redundant data such as air speed or fuel levels. We intend to extend this work to incorporate quantitative logical inference based on spatio-temporal knowledge and constraints to promote autonomous data stream management. A method for enforcing logical constraints within streaming applications is presented in [12]. A comprehensive look at the variations of spatio-temporal logic and their computational complexity is presented in [13]: the dichotomy of qualitative and quantitative logic is discussed with respect to space and time. Further research on spatio-temporal logic and constraint logic programming includes [14, 15].

Acknowledgments

This research is partially supported by the Air Force Office of Scientific Research Grant No. FA9550-11-1-0332.

References

[1] B. d'Enquêtes et d'Analyses pour la Sécurité de l'Aviation Civile, Final Report: On the accident on 1st June 2009 to the Airbus A330-203 registered F-GZCP operated by Air France flight AF 447 Rio de Janeiro - Paris.
URL `http://e1.flightcdn.com/live/special/Air_France_447_AFR447_Final_Report-en.pdf`

[2] S. Imai, C. A. Varela, Programming spatio-temporal data streaming applications with high-level specifications, in: 3rd ACM SIGSPATIAL International Workshop on Querying and Mining Uncertain Spatio-Temporal Data (QUeST) 2012, Redondo Beach, California, USA, 2012.

[3] S. Imai, C. A. Varela, A programming model for spatio-temporal data streaming applications, in: Dynamic Data-Driven Application Systems (DDDAS 2012), Omaha, Nebraska, 2012, pp. 1139–1148.

[4] K. An, J. Kim, Moving objects management system supporting location data stream, in: Proceedings of the 4th WSEAS international conference on Computational intelligence, man-machine systems and cybernetics, CIMMACS'05, World Scientific and Engineering Academy and Society (WSEAS), Stevens Point, Wisconsin, USA, 2005, pp. 99–104.

[5] M. Stonebraker, U. Çetintemel, S. Zdonik, The 8 requirements of real-time stream processing, SIGMOD Rec. 34 (4) (2005) 42–47.

[6] B. Babcock, S. Babu, M. Datar, R. Motwani, J. Widom, Models and issues in data stream systems, in: Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems, PODS '02, ACM, New York, NY, USA, 2002, pp. 1–16.

[7] D. J. Abadi, Y. Ahmad, M. Balazinska, M. Cherniack, J. hyon Hwang, W. Lindner, A. S. Maskey, E. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, S. Zdonik, The design of the borealis stream processing engine, in: In CIDR, 2005, pp. 277–289.

[8] A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, K. Ito, R. Motwani, U. Srivastava, J. Widom, Stream: The stanford data stream management system, Springer, 2004.

[9] B. Gedik, H. Andrade, K.-L. Wu, P. S. Yu, M. Doo, Spade: the system s declarative stream processing engine, in: Proceedings of the 2008 ACM SIGMOD international conference on Management of data, SIGMOD '08, ACM, New York, NY, USA, 2008, pp. 1123–1134.

[10] G. Jacques-Silva, B. Gedik, R. Wagle, K.-L. Wu, V. Kumar, Building user-defined runtime adaptation routines for stream processing applications, Proc. VLDB Endow. 5 (12) (2012) 1826–1837.

[11] M. Branson, F. Douglis, B. Fawcett, Z. Liu, A. Riabov, F. Ye, Clasp: collaborating, autonomous stream processing systems, in: Proceedings of the ACM/IFIP/USENIX 2007 International Conference on Middleware, Middleware '07, Springer-Verlag New York, Inc., New York, NY, USA, 2007, pp. 348–367.

[12] A. Lallouet, Y.-C. Law, J. H.-M. Lee, C. F. K. Siu, Constraint programming on infinite data streams., in: T. Walsh (Ed.), IJCAI, IJCAI/AAAI, 2011, pp. 597–604.

[13] F. Wolter, M. Zakharyaschev, Qualitative spatio-temporal representation and reasoning: A computational perspective, in: Exploring Artifitial Intelligence in the New Millenium, Morgan Kaufmann, 2001, pp. 175–216.

[14] R. Gennari, Temporal reasoning and constraint programming - a survey, CWI Quarterly 11 (1998) 3–163.

[15] A. Raffaeta, T. W. Fralhwirth, Spatio-temporal annotated constraint logic programming., in: PADL'01, 2001, pp. 259–273.

Programming Spatio-Temporal Data Streaming Applications with High-Level Specifications

Shigeru Imai
Department of Computer Science
Rensselaer Polytechnic Institute
110 Eighth Street
Troy, NY 12180, USA
imais@cs.rpi.edu

Carlos A. Varela
Department of Computer Science
Rensselaer Polytechnic Institute
110 Eighth Street
Troy, NY 12180, USA
cvarela@cs.rpi.edu

ABSTRACT

In this paper, we describe the design and implementation of PILOTS, a ProgrammIng Language for spatiO-Temporal data Streaming applications. Using PILOTS, application developers can easily program an application that handles spatio-temporal data streams by writing a high-level declarative program specification.

Whereas spatio-temporal data is available with various spatial density and time frequency depending on data sources (*e.g.*, weather forecast data can be given hourly/-daily for a vast geographic area, while GPS data can be given every second or at a higher frequency for a specific geographic location), applications often need to process data at a constant frequency. To view such heterogeneous data streams as *homogeneous* data streams, PILOTS specifically provides first-class support for data selection and interpolation so that applications can get data consistently regardless of the data's original spatio-temporal heterogeneity.

To enable reasoning about errors in correlated spatio-temporal data streams, we introduce the notion of *error signatures*, patterns in output data streams that appear when input data is erroneous. These patterns are produced thanks to a mathematical model that explicitly specifies the redundancy exhibited in the input data. PILOTS applications readily produce error signatures, which can be an important tool to semi-automatically detect data error conditions and enable better decision support systems.

As a motivating application, we illustrate a PILOTS program that receives as input data: the airspeed, the ground speed, and the wind speed for a flight. We then compute the error signatures exhibited by failing the airspeed data stream simulating a pitot tube icing scenario (such as the one occurring in Air France flight 447 in June 2009 ultimately killing all people onboard), and by failing the ground speed data stream simulating a GPS constellation shutdown.

ACM SIGSPATIAL QUeST'12 November 6, 2012. Redondo Beach, CA, USA

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features—*Frameworks*

General Terms

Languages

Keywords

programming language, spatio-temporal, data streaming

1. MOTIVATION

Operating an aircraft is known as a complicated task since there are a lot of complex correlations between the readings in a cockpit's instruments. If a failure happens during a flight, it is not easy to find the cause of the failure by looking at the available (potentially partially erroneous) data, and also, a misinterpretation of instrument readings could even lead to an accident as the tragic crash of Air France flight 447, killing all 216 passengers and 12 aircrew [10]. The records from the crash have suggested that the pilots lost control of the airplane because they raised the nose of the airplane when it should not have been brought up. Many experts now understand that the airplane went into clouds with thunderstorms and its iced speed sensors provided inaccurate information to the autopilot, causing it to disengage. The pilots then incorrectly reacted to the emergency by raising the nose of the plane when in fact it needed to go down to avoid the stall.

An active redundant data-driven flight system may help prevent crashes caused by malfunctioning sensors or other data errors. For example, by comparing the air speed data to the ground speed data, a flight system would be able to fact check a bad air speed reading, assuming reasonable constraints on the wind speed. If the (auto) pilot is only operating by air speed data, they would have no way of knowing that there is an error in the system and they would respond to the incorrect data, upsetting the balance of the plane. The ground speed data would instead provide a fact checking mechanism because if airspeed were swiftly changing, ground speed would be doing the same. If airspeed is changing, but ground speed remains unchanged, the more active flight system would be able to notify the pilot of the discrepancy, allowing for better informed decision making.

Considering the development of such an active flight system, applications running on the flight system should deal

with (1) the airplane's constantly changing location and potentially inaccurate and incomplete input data streams from various sensors and (2) reasoning about the input data streams to identify failures and their potential sources using redundancy among the input data streams.

To provide a technological foundation towards the purpose of (1), we have defined a programming model for spatio-temporal data streaming applications [3], which is specifically designed for moving objects (*e.g.,* airplanes, cars, trains, and so on) to take spatio-temporal data streams as inputs and output processed data streams. In this paper, we describe the design and implementation of a programming language following the model: *PILOTS* (ProgrammIng Language for spatiO-Temporal data Streaming applications). PILOTS provides first-class support for space and time specific operations including data selection and interpolation when no data is available for a certain location and time. We also show some experimental results of running PILOTS code on actual flight data with simulated error conditions to produce error signatures, an important step towards the realization of (2).

The rest of the paper is organized as follows. Section 2 introduces the PILOTS programming language for spatio-temporal data streaming applications. Section 3 describes the implementation of the language in detail, and Section 4 presents experimental results of running PILOTS code. Section 5 shows related work, and finally we conclude the paper in Section 6 highlighting potential future work.

2. SPATIO-TEMPORAL PROGRAMMING LANGUAGE

2.1 System Architecture

Figure 1 shows the system architecture for applications implemented in the PILOTS programming language. In this architecture, the application gets data $(d'_1, d'_2, \ldots, d'_n)$ from the *Data Selection* module, which takes incoming data streams $(d_1, d_2, \ldots, d_n)$ as inputs, and then the application indefinitely generates outputs $(o_1, o_2, \ldots, o_m)$ and data errors $(e_1, e_2, \ldots, e_l)$ based on an *Application Model*. Each input data stream $d_i(x, y, z, t)$ is a function of location and time. The number of arguments of d_i varies depending on dimensions of the location information, that is, $d_i(t)$ for 0-D (*i.e.*, no location support), $d_i(x, t)$ for 1-D, $d_i(x, y, t)$ for 2-D, and $d_i(x, y, z, t)$ for 3-D. Some data streams are coming in real-time whereas some predicted information (*e.g.*, weather forecasts) is associated with future time periods.

The Data Selection module stores some amount of incoming data stream until it becomes out of date. The application acquires the selected or interpolated data $(d'_1, d'_2, \ldots, d'_n)$ from the Data Selection module at a certain rate specified in the Application Model and computes both outputs and data errors. The application continues this computing process in an infinite loop unless the user explicitly specifies the termination time.

Whereas spatio-temporal data is available with various spatial density and time frequency depending on sources of data in general (*e.g.*, weather forecast data can be given hourly/daily for a vast geographic area, GPS data can be given every second or at higher frequency for a specific geographic location), the application often needs to process data at a constant frequency. The Data Selection module essentially allows an application to view a set of these heterogeneous data streams as a homogeneous data stream, and therefore enables a separation of concerns: application programmers can focus on their application model.

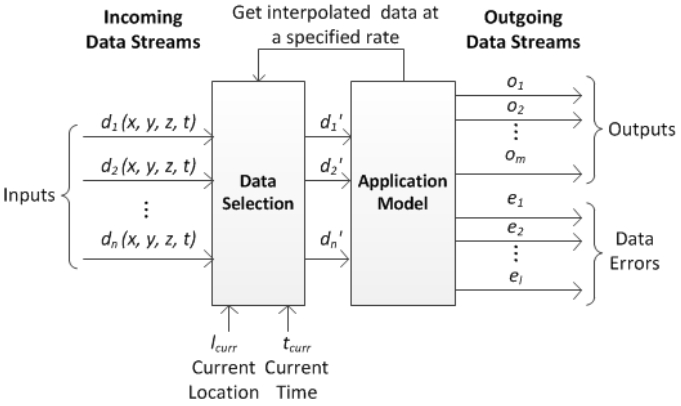


Figure 1: System architecture of PILOTS programs which handle spatio-temporal data streams

2.2 First Class Support for Data Selection

PILOTS specifically implements first-class support for data selection and interpolation in the Data Selection module so that the application can get data consistently regardless of its heterogeneity. Here we explain three methods for data selection and interpolation: *closest, euclidean*, and *interpolate* and show an example use of these methods.

2.2.1 Data Selection/Interpolation Methods

- *closest*
  This method takes a 1-D argument (*i.e.,* t, x, y, or z) to find the data closest to a given location or time. Figure 2 shows examples of selecting closest data to the current time and location respectively. In Figure 2(a), when selecting the closest time to the current time t_{curr}, $d_i(t_{curr})$ is not defined, but $d_i(t)$ is defined for $\{t \mid t_1 \leq t \leq t_2, t_3 \leq t \leq t_4, t_5 \leq t \leq t_6\}$. Since t_4 is closest to t_{curr}, we define $d'_i(t_{curr}) \triangleq d_i(t_4)$. Similarly, we define $d'_i(x_{curr}) \triangleq d_i(x_3)$ for the example shown in Figure 2(b).

- *euclidean*
  This method takes 2-D or 3-D arguments to find the data closest to a given location. Figure 3 shows an example for the 2-D case, where data is not defined for the current location $l_{curr} = (x_{curr}, y_{curr})$, but are defined for l_0, and l_1. Since l_{curr} is closest to $l_0 = (x_0, y_0)$ in Euclidean distance, we define $d'_i(x_{curr}, y_{curr}) \triangleq d_i(x_0, y_0)$.

- *interpolate*
  This method takes 1-D, 2-D or 3-D arguments to linearly interpolate the defined data. It also takes another argument n_{interp} to select the closest n_{interp} data from a given location to interpolate. Suppose we have a situation shown in Figure 4, where data is not defined for the current location $l_{curr} = (x_{curr}, y_{curr})$, but are defined for l_0, l_1, and l_2. Also, suppose that n_{interp} is

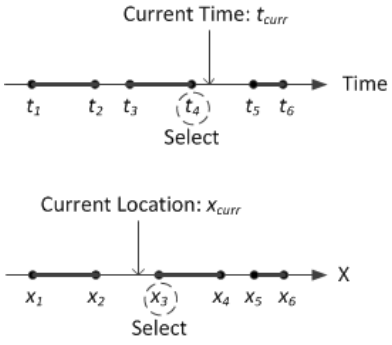


Figure 2: (a) Selecting the closest time (above); (b) Selecting the closest x value (below)

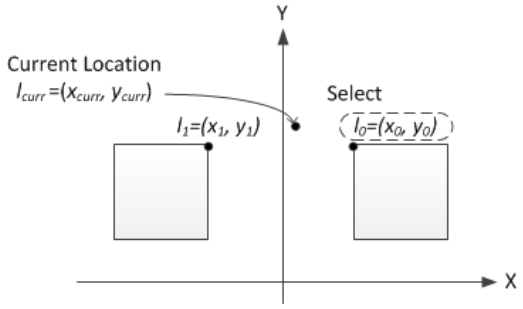


Figure 3: Selecting the closest 2D region in Euclidean distance

2, we select l_0 and l_1 since they are closer to l_{curr} than l_2. In such a case, we linearly interpolate the data defined for l_0 and l_1 by taking a weighted sum based on the Euclidean distance as follows:

$$d'_i(x_{curr}, y_{curr}) \triangleq (1 - \frac{\|l_0 - l_{curr}\|}{\sum_{j=0}^{1} \|l_j - l_{curr}\|}) \cdot d_i(x_0, y_0) + (1 - \frac{\|l_1 - l_{curr}\|}{\sum_{j=0}^{1} \|l_j - l_{curr}\|}) \cdot d_i(x_1, y_1) \quad (1)$$

Note that the equation (1) can be easily extended to n data points.

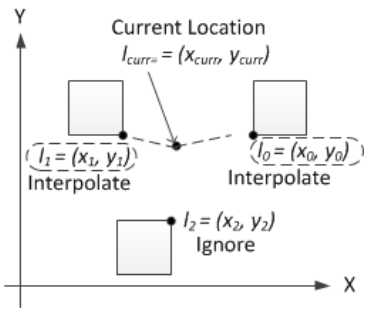


Figure 4: Linear interpolation

Table 1: Wind speed prediction information (ID:0-2 for Albany, ID:3-5 for Pittston, and ID:6-8 for JFK Airport

ID	latitude(x), longitude(y)	altitude(z) [ft.]	time(t)	windspeed [knot]
0	42.73,-73.69	3000	04/03/12 14:00	20
1	42.73,-73.69	6000	04/03/12 14:00	32
2	42.73,-73.69	9000	04/03/12 14:00	40
3	41.34,-75.72	3000	04/03/12 14:00	17
4	41.34,-75.72	6000	04/03/12 14:00	31
5	41.34,-75.72	9000	04/03/12 14:00	41
6	40.64,-73.78	3000	04/03/12 14:00	18
7	40.64,-73.78	6000	04/03/12 14:00	33
8	40.64,-73.78	9000	04/03/12 14:00	43

2.2.2 *Example*

Imagine you are flying in an airplane at the altitude of 7000 ft. in the middle of New York state as shown in Figure 5. Given the predicted wind speed information for Albany, Pittston, and JFK Airport in Table 1, how can we estimate a reasonable wind speed for the current location at the current time?

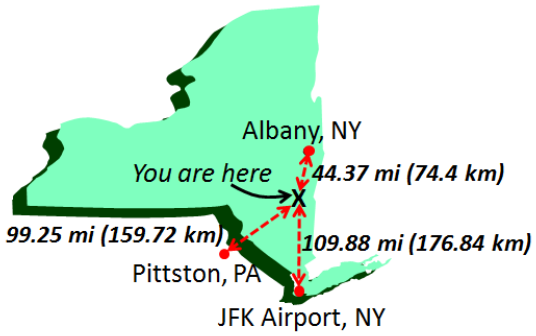


Figure 5: Example geographical relationship between the current location and surrounding cities

Suppose the current location represented by (latitude, longitude, altitude) is (42.20, -74.18, 7000 ft.) and the current time is 04/03/12 15:34, we can get a wind speed by applying `euclidean(x,y)`, `closest(t)`, and `interpolate(z,2)` sequentially as follows.

1. Apply `euclidean(x,y)` to all the data. Looking at latitude(x) and longitude(y), the closest city to the current location is Albany as shown in Figure 5. Select the data from ID:0 to ID:2.

2. Apply `closest(t)` to data ID:0-2. Looking at time(t), the current time is equally close to the time of data ID:0-2. Select all three data.

3. Apply `interpolate(z,2)` to data ID:0-2. Since n_{interp} is 2, pick up the two closest data in altitude(z), which are ID:1 and 2. Finally, calculate the final wind speed value similar to the equation (1) as $(1 - 1000/3000) * 32 + (1 - 2000/3000) * 40 = 13.33 + 21.33 = 34.66$ [knot].

2.3 Example Program Specifications

Here we show two example program specifications; one is very simple and the other is slightly more complex than the first one. See [3] for the detailed PILOTS grammar definition. We will see the experimental results of these programs later in Section 4.

2.3.1 Simple Example - Twice

Figure 6 shows one of the simplest program specifications written in PILOTS, called `twice`. As the name says, it takes two input streams, $a(t)$ and $b(t)$, where b is supposed to be twice as large as a, and outputs an error defined by $e = (b - 2 * a)$ every 1 second. Note that these two input streams are not associated with any location information.

```
program twice;
    inputs
        a: (t) using closest(t);
        b: (t) using closest(t);
    outputs;
    errors
        e: (b - 2 * a) at every 1 sec;
end;
```

Figure 6: A simple declarative specification of the twice application

2.3.2 More Complex Example - Flight Planning

This is an example of a simplified flight planning system. Suppose that sensors in an airplane record airspeeds v_a during a given flight and GPS units record the airplane's flight path over the ground including ground speeds v_g at different locations. An aircraft's airspeed and ground speed are related by the following mathematical formula (2), where v_a and α_a are the aircraft airspeed and angle (heading), and v_w and α_w are the wind speed and direction acquired from the weather forecast:

$$v_g = \sqrt{v_a^2 + 2v_a \cdot v_w \cdot \cos(\alpha_a - \alpha_w) + v_w^2} \quad (2)$$

Also, we can compute crosswind velocity: $v_x = v_w \cdot \sin(\alpha_a - \alpha_w)$. Therefore, given the aircraft desired course α_d, it is possible to compute the crab angle δ by using the formula (3) so that the aircraft can use $\alpha_a = \alpha_d + \delta$ as the heading to maintain the desired direction under varying wind conditions.

$$\begin{aligned} \delta &= \arcsin(v_x / v_g) \\ &= \arcsin\left(\frac{v_w \cdot \sin(\alpha_a - \alpha_w)}{\sqrt{v_a^2 + 2v_a \cdot v_w \cdot \cos(\alpha_a - \alpha_w) + v_w^2}}\right) \end{aligned} \quad (3)$$

The above mentioned relationship can be brought into a program specification shown in Figure 7, which outputs the crab angle δ and error e that is the difference between the monitored ground speed v_g from GPS and the calculated one with the equation (2).

In this flight planning example, there are three input data streams in which each stream has two functions. Since each of these two functions has the same source of information and arguments, they are declared as a single input data stream. In the case of the first input data stream, there are two functions, `wind_speed(x,y,z,t)` and `wind_angle(x,y,z,t)`, that share the same arguments and information source (weather forecast). Data interpolation/selection methods used for these two functions are `euclidean(x,y)`, `closest(t)`, and `interpolate(z,2)`. Just like we explain in Section 2.2.2, these methods apply in order: first, the closest x and y to x_{curr} and y_{curr} in Euclidean distance are selected; second, the closest t to the current time t_{curr} is selected; and finally, the final value is linearly interpolated on the z-axis using up to the two closest data points to z_{curr} as specified in the argument.

```
program flightplan;
    inputs
        wind_speed, wind_angle: (x,y,z,t)
            using euclidean(x,y), closest(t),
            interpolate(z,2);

        air_speed, air_angle: (x,y,t)
            using euclidean(x,y), closest(t);

        ground_speed, ground_angle: (x,y,t)
            using euclidean(x,y), closest(t);

    outputs
        crab_angle:
            arcsin(wind_speed *
                sin(wind_angle-air_angle) /
                sqrt(air_speed^2 +
                  2*air_speed*wind_speed *
                  cos(air_angle-wind_angle) +
                  wind_speed^2))
            at every 1 min;

    errors
        e: ground_speed -
            sqrt(air_speed^2 +
                2*air_speed*wind_speed *
                cos(air_angle-wind_angle) +
                wind_speed^2)
        at every 1 min;
end;
```

Figure 7: A declarative specification of the flight planning application

3. LANGUAGE IMPLEMENTATION

When executing a PILOTS program, its high-level program specification needs to be compiled into Java code by the PILOTS compiler. The generated application program then uses the PILOTS runtime library to run the program.

3.1 Compiler

The compiler consists of two parts: a parser and a code generator. The parser is developed using JavaCC [4]. Then the code generator uses the abstract syntax tree created from the parser and applies visitor pattern to generate Java code.

3.2 System Interaction

Figure 8 shows how a PILOTS application interacts with the system. The interactions are based on a client-server model using Internet sockets in which the application works as a server and takes inputs from the clients on a single port.

It outputs some values to the specified output ports as well as error values to the specified error ports.

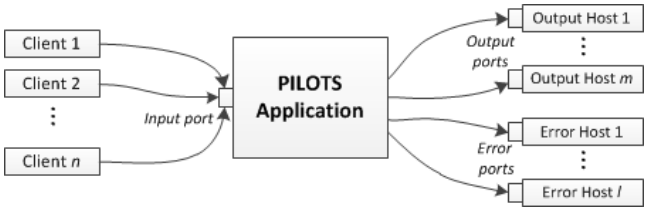


Figure 8: System interaction of a PILOTS application

When executing a compiled Java program, we specify input, output, and error ports as follows. In this example, the `flightplan` application illustrated in Section 2.3.2 takes an input data stream on the port `10001` and sends output and error streams to the hosts specified by `10.0.0.1:20001` and `10.0.0.2:30001` respectively.

```
$ java pilots.tests.Flightplan -input 10001
-outputs 10.0.0.1:20001 -errors 10.0.0.2:30001
```

3.2.1 Data Format

The input, output, and error data streams share the same format shown in Table 2. The first line is used to declare one or more variables (`var0, var1,...`) in a single data stream. The values of the declared variables start from the second line. The data stream can have multiple values (`val0, val1,...`) with various spatial and temporal combinations: ex1 is just defined for a 2-D region; ex2 is defined for a 3-D point and a time interval; ex3 is defined for a 1-D interval and a particular time; ex4 is defined for no location and a particular time. All lines have to end with an end-of-line marker (`\n`). Especially, the last line has to have only one end-of-line marker.

Note that the data format for input is compatible with output and error, that is, we can connect either an output or error port to an input port of another PILOTS application.

Table 2: The data stream format of input, output, and error

first line	`#var0,var1,...\n`
after second line	ex1) `x0,y0~x1,y1::val0,val1,...\n` ex2) `x,y,z:t0~t1:val0,val1,...\n` ex3) `x0~x1:t:val0,val1,...\n` ex4) `:t:val0,val1,...\n`
last line	`\n`

Here are some instances of spatio-temporal data for ex1$\cdots$ex4 respectively.

- ex1) 40.100,-76.300~39.600,-76.300::166.0,215.0
- ex2) 42.749,-73.802,3000:2012-04-03 140000-0500 ~2012-04-03 210000-0500:15.0,320.0
- ex3) 42.6886~43.9258:2012-04-03 140900-0500 :112.0,222.0
- ex4) :2012-04-03 141400-0500:42.5486,-74.1142,8100.0

3.2.2 Running Mode

Two modes are available for the user to run PILOTS applications as shown below.

- *real-time mode:* This mode is default and is intended to be used for receiving data from sensors and processing it in real-time. If the frequency of an output is specified as "`at every 1 min`" in the program specification, the program actually outputs data once every 1 minute. Also, in this mode, the program finishes if one of input data streams sends the last line marker (`\n`) or certain amount of time has elapsed, which can be specified by the user in the command line as `-DtimeSpan=30min`.
- *simulation mode:* This mode is used for simulations and is activated if the user gives a past time span in the command line as `-DtimeSpan=t0~t1`. The PILOTS runtime sets its internal time as `t0` and virtually progress the time as the program runs, and when the internal time reaches `t1`, the program finishes. The user can get outputs as fast as possible. This mode is intended to be used for processing recorded data in the past.

In either mode, time stamp of input streams should match the internal time of the PILOTS runtime to get proper outputs.

3.3 Runtime Library

The PILOTS runtime library is in charge of starting a data receiving server, storing received data, providing data selection/interpolation service to the application, and sending processed data to output/error hosts.

Primary classes included in the PILOTS runtime library shown in Figure 9 are explained as follows.

- `PilotsRuntime` class is extended by the application and provides all basic functions to run a PILOTS application other than application-specific processing. It starts `DataReceiver` to start receiving data, requests stored data from `DataStore`, and sends calculated outputs and errors to other hosts.
- `DataReceiver` class receives data from data input clients from a port specified in the command-line arguments. Upon accepting data, it launches a new worker thread to receive data and the created thread requests to add these data to `DataStore`.
- `DataStore` class accepts data from `DataReceiver` as a string, and then it asks `SpatioTempoData` to parse the string and stores the parsed data. It also implements `getData()` method supporting `closest, euclidean, interpolate` for data selection. When comparing locations and time for data selection/interpolation, it asks for the current time and location from `CurrentLocationTimeService`. Stored data are accessed from multiple threads (*i.e.*, threads for adding data from `DataReceiver` vs. threads getting data from `PilotsRuntime`), so the data have to be protected from simultaneous data access.
- `CurrentLocationTimeService` class is an interface class for providing the current time and location. Users have to implement this class for the system to work

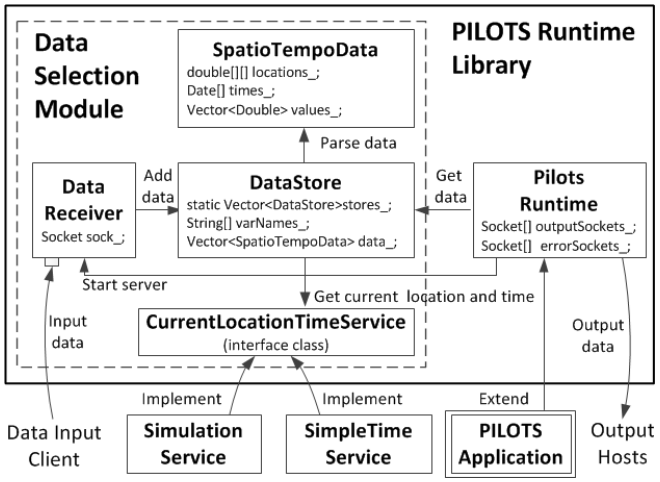


Figure 9: Class diagram of PILOTS runtime library

(*e.g.*, `SimpleTimeService` and `SimulationService`). The implemented class can either return the actual current time for the real-time mode or past time for the simulation mode.

4. EXPERIMENTS

In this section, we report the results of two experiments. The first experiment has been done with the `twice` application to illustrate how error signatures exhibit different shapes depending on simulated input data streams. The second one has been conducted with the `flightplan` application to apply PILOTS to data sampled in the real world and see how PILOTS' first-class support for data selection and interpolation work for real data.

4.1 Simulated Data Inputs - Twice

4.1.1 Experimental Setup

As we present in Figure 6, the `twice` application takes two inputs, `a` and `b`, and we prepare an independent client for each input to test the following four different scenarios.

- Scenario A: There are random timing jitters between one data and another on the data input clients. That is, every $1 \pm \epsilon$ seconds, the variable `a`'s input comes as $1, 2, 3, \ldots$, whereas the variable `b`'s input comes as $2, 4, 6, \ldots$.
- Scenario B: The variable `a`'s data stream becomes consistently one second behind the variable `b`'s input data stream at some point of time.
- Scenario C: The variable `a`'s data stream stops providing data at some point of time.
- Scenario D: The variable `b`'s data stream stops providing data at some point of time.

In all the scenarios, the application runs in the real-time mode with `SimpleTimeService` that returns the current time only. We start the `twice` application followed by the data input clients, and then record the error output for 120 seconds once data from the clients reaches the PILOTS runtime.

4.1.2 Results

Figure 10 shows different types of errors generated with the four different scenarios. In Figure 10(a), most of the time the error stays at zero, but there are several spikes due to transient fluctuation of the data input timing. It happens occasionally since the use of `closest(t)` causes the Data Selection module to select data at one second earlier or one second later than it is supposed to select. This type of error is unavoidable without a special synchronization mechanism between multiple data streams. Figure 10(b) shows a signature of out-of-sync input data streams. As shown in the graph, the error becomes consistently large at around 30 seconds of the simulation time. This is because the variable `a`'s input data stream becomes consistently one second behind the variable `b`'s input data stream. Figure 10(c) suggests more critical failure of the variable `a`'s input data source. At around 50 seconds of the simulation time, the error starts growing linearly. This linear increase of the error explains that the input data stream of the variable `a` stops coming after 50 seconds of the simulation time, which potentially means that a critical failure occurred at the source of the variable `a`. Similarly, Figure 10(d) suggests that a critical failure occurred at the source of the variable `b`.

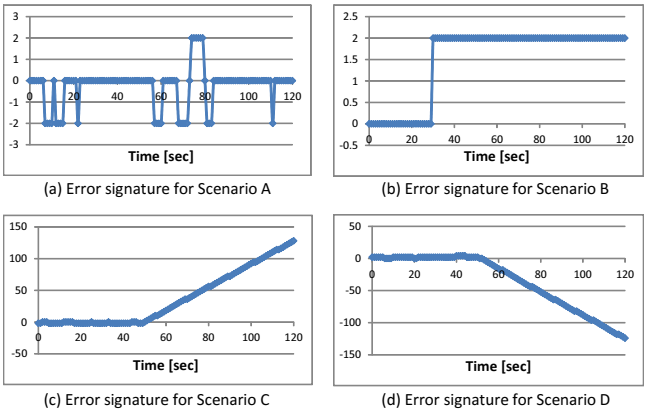


Figure 10: Error signatures generated by the `twice` application

As we can see from the graphs, errors behave differently depending on the input data streams, thus those error signatures could tell us valuable information about potential failures in the data sources.

4.2 Real Data Inputs - Flight Planning

4.2.1 Experimental Setup

The second author is a general aviation private pilot and we conducted a simulation with the `flightplan` program based on his actual flight from Albany, New York to Tipton, Maryland (near Washington D.C.) on April 3rd, 2012. `air_speed` and `air_angle` were manually collected during the flight, whereas `ground_speed` and `ground_angle` were automatically collected from online data [2]. We have to use weather forecast information for `wind_speed` and `wind_angle` from [6] since we could not record the actual wind information during the flight.

Note that `air_speed` and `air_angle` are sparsely given as shown in Figure 11 since they were manually recorded while

operating the aircraft, and also the density of `wind_speed` and `wind_angle` is not very high as we have previously seen in Table 1. Unlike other data, `ground_speed` and `ground_angle` are given every minute.

```
#air_speed,air_angle
42.748,-73.802~41.476,-75.483::162,246
41.476,-75.483~41.000,-76.000::161,239
41.000,-76.000~40.100,-76.300::161,221
40.100,-76.300~39.600,-76.300::166,215
39.600,-76.300~39.500,-76.400::165,213
39.500,-76.400~39.085,-76.759::135,204
```

Figure 11: Formatted data for airspeed and air angle from the April 3rd flight

Just like the previous section with the `twice` application, we test the following four different scenarios including three simulated error conditions.

- Scenario A: No error is added. Use the real data only.
- Scenario B: Simulate an airspeed sensor (called pitot tube) icing failure. If the airspeed sensor is iced, typically the airspeed suddenly drops in a few seconds, and then it keeps reporting a constant value.
- Scenario C: Simulate a GPS failure. GPS loses satellites and keeps reporting 0s as the values for both ground speed and ground angle.
- Scenario D: Simulate both an airspeed sensor icing failure and a GPS failure mentioned above.

For all the scenarios, we run the program in the simulation mode with `SimulationService` that returns the location and time based on the actual flight path. To specify the start time and end time of the simulation, we run the program with the option `-DtimeSpan="2012-04-03 1404~2012-04-03 1545"` for the 1 hour and 41 minutes flight.

4.2.2 Results

Figure 12 plots the error and crab angle from the flight planning application for Scenario A (no errors). Looking at the graph, we notice that the error is large at the beginning of the flight (around 0~9 minutes) and also at the end of the flight (around 90~100 minutes). The reason is inaccuracy of the airspeed, which is illustrated in Figure 13(a) in which we plot the airspeed, ground speed, calculated ground speed, and error separately from the outputs of the `flightplan` application to analyze the reasons of the error. In the graph, we see that the airspeed is almost constant whereas actual ground speed changes dynamically at the time of departure and landing. In general, airspeed is supposed to change a lot when departing and landing; however, since airspeed was manually recorded and these periods are the busiest for flying the airplane, it is not accurate. Consequently, inaccuracy in airspeed makes the calculated ground speed erroneous since it is directly related to the airspeed according to the equation (2). Despite the large error when departing and landing, the error stays relatively low around 10~90 minutes, and that fact suggests PILOTS' data selection/interpolation methods work well during this period since the relationship presented by the equation (2) holds well.

In the case of Scenario B, Figure 13(b) shows that the error suddenly increases at around 40 minutes caused by a sudden drop of the airspeed (150 to 50 knots in two minutes). Unlike Scenario B, Scenario C shows a different error signature as shown in Figure 13(c). From the graph, we can see that the ground speed suddenly drops from 170 to 0 knot at around 40 minutes and that causes the error to drop accordingly. When we have both failures, we get an error signature as shown in Figure 13(d). This error signature tells us that the errors for Scenario B and C do not cancel out, but they are emerged as a combination of the individual error signatures. This result potentially means that we may be able to tell the causes of multiple errors from one combined error signature.

As we can see from the graphs, there is a clear distinction between the error signatures. These results encourage us to pursue automated reasoning about data errors and recovery.

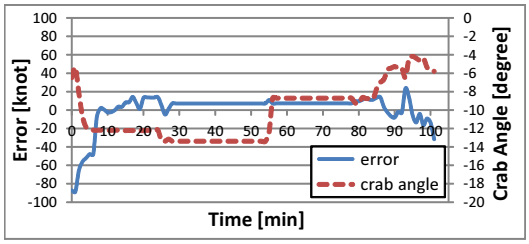


Figure 12: Error and output generated by the `flightplan` application

5. RELATED WORK

Spatio-temporal constraint logic programming has been proposed. STACLP [7] offers first-class support for representing and reasoning about spatial and temporal data. A similar logic language to STACLP, MuTACLP [5][1], has been applied to GIS for spatio-temporal reasoning [8]. Both STACLP and MuTACLP are implemented based on a Prolog system. Programming languages that support probabilistic reasoning have also been proposed. PRISM[9] is a logic-based language that integrates logic programming and stochastic reasoning including parameter learning. PRISM is capable of parameter learning from a given set of data and estimates the probability to best explain the data. PRISM is also built on top of a Prolog system. Our programming language is also highly declarative and generates code to help understand and reason about spatio-temporal data streams.

6. CONCLUSIONS

We presented the design and implementation of PILOTS, a programming language for spatio-temporal data streaming applications. The language enables to specify in a declarative (high-level) manner the mathematical relationship between data streams. We also illustrated different methods that can be combined to interpolate and select the data. These methods enable developers to declaratively specify how to convert potentially *heterogeneous* data streams—*i.e.*, streams using different scales in space and time—into *homogeneous* data streams amenable to processing and analysis.

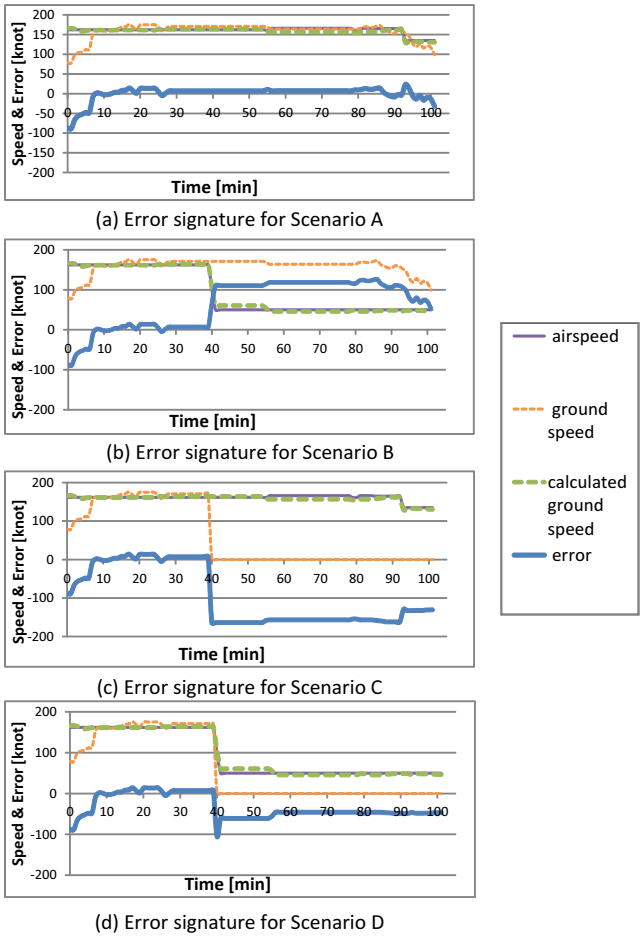


Figure 13: Error signatures and speed information generated by the `flightplan` application

An important goal of this work is to enable spatio-temporal data analyses that detect errors in input data streams with high probability. This is accomplished by explicitly modeling redundancy in the input data streams allowing application developers to not only specify output data streams but also to specify mathematically the relationship between redundant data as *error streams*. These error streams can be seen as *signatures* that characterize different input data error conditions with high probability.

We used PILOTS to uncover the error signatures of a trivial application (*twice*) illustrating clearly distinguishable patterns on different kinds of failure (out of synchronization and data stream loss failures). We subsequently used PILOTS to model the relationship between air speed, ground speed, and wind speed of an actual general aviation flight between Albany, NY and Washington, DC. We computed the error signatures for (*i*) normal conditions, (*ii*) a simulated pitot tube failure (affecting the airspeed data stream), (*iii*) a simulated GPS constellation failure (affecting the ground speed data and ground angle stream,) and (*iv*) a simultaneous failure of the pitot tube (*ii*) and GPS system (*iii*). These error signatures illustrate that patterns can be used to discover with high likelihood the source of potential data errors.

Future work includes modeling more complex data relationships in aviation and navigation systems, discovering error signatures for common data error conditions, using error signatures as a means to semi-automate error recovery and reason about spatio-temporal data streams, and applying the programming model and language to other domains generalizing it as appropriate.

ACKNOWLEDGMENTS

We would like to thank Jaemyeong Eo and Yazmin Feliz for their help on creating test data. This research is partially supported by Air Force Office of Scientific Research Grant No. FA9550-11-1-0332.

7. REFERENCES

[1] P. Baldan, P. Mancarella, A. Raffaetà, and F. Turini. MuTACLP: A language for temporal reasoning with multiple theories. In *Computational Logic: Logic Programming and Beyond'02*, pages 1–40, 2002.

[2] FlightAware. Flight track log for N756VH on 03-Apr-2012 (KALB-KFME). `http://flightaware.com/live/flight/N756VH/history/20120403/1800Z/KALB/KFME/tracklog`.

[3] S. Imai and C. A. Varela. A programming model for spatio-temporal data streaming applications. In *Dynamic Data-Driven Application Systems (DDDAS 2012)*, pages 1139–1148, Omaha, Nebraska, June 2012.

[4] JavaCC Development Group (Open Source Project under BSD License). Java compiler compiler (javacc) - the java parser generator. `http://javacc.java.net/`.

[5] P. Mancarella, G. Nerbini, A. Raffaetà, and F. Turini. MuTACLP: A language for declarative GIS analysis. In *Computational Logic'00*, pages 1002–1016, 2000.

[6] NOAA's National Weather Service. Forecast winds and temps aloft. `http://aviationweather.gov/products/nws/winds/`.

[7] A. Raffaetà and T. W. Frühwirth. Spatio-temporal annotated constraint logic programming. In *PADL'01*, pages 259–273, 2001.

[8] A. Raffaetà, F. Turini, and C. Renso. Enhancing GISs for spatio-temporal reasoning. In *Proceedings of the 10th ACM international symposium on Advances in geographic information systems*, GIS '02, pages 42–48, New York, NY, USA, 2002. ACM.

[9] T. Sato and Y. Kameya. Parameter learning of logic programs for symbolic-statistical modeling. In *Journal of Artificial Intelligence Research(JAIR)*, volume 15, pages 391–454, 2001.

[10] Wikipedia. Air France Flight 447. `http://en.wikipedia.org/wiki/Air_France_Flight_447`.

Available online at www.sciencedirect.com
ScienceDirect
Procedia Computer Science 00 (2012) 1–10
Procedia Computer Science

International Conference on Computational Science, ICCS 2012

A programming model for spatio-temporal data streaming applications

Shigeru Imai[a,*], Carlos A. Varela[a]

[a] *Computer Science Department, Rensselaer Polytechnic Institute, 110 Eighth Street, Troy, NY 12180, USA*

Abstract

In this paper, we describe a programming model to enable reasoning about spatio-temporal data streams. A spatio-temporal data stream is one where each datum is related to a point in space and time. For example, sensors in a plane record airspeeds (v_a) during a given flight. Similarly, GPS units record an airplane's flight path over the ground including ground speeds (v_g) at different locations. An aircraft's airspeed and ground speed are related by the following mathematical formula: $v_g = \sqrt{v_a^2 + 2v_a \cdot v_w \cdot \cos(\alpha_a - \alpha_w) + v_w^2}$, where v_a and α_a are the aircraft airspeed and heading, and v_w and α_w are the wind speed and direction. Wind speeds and directions are typically forecast in 3,000-foot height intervals over discretely located fix points in 6-12 hour ranges. Modeling the relationship between these spatio-temporal data streams allows us to estimate with high probability the likelihood of sensor failures and consequent erroneous data. Tragic airplane accidents (such as Air France's Flight 447 on June 1st, 2009 killing all 216 passengers and 12 aircrew aboard) could have been avoided by giving pilots better information which can be derived from inferring stochastic knowledge about spatio-temporal data streams. This work is a first step in this direction.

Keywords: programming models, spatio-temporal data, data streaming

1. Introduction

Spatio-temporal data streams, where each datum is related to a point or a range of space and time, are pervasive. We see such data streams in many occasions in our daily lives, for instance, temperatures, prices of gasoline, flight schedules of airplanes, and so on. Massive amounts of spatio-temporal data are continuously generated by sensors, IT systems, or computer programs, and consumed by humans; however, today's most commonly used programming languages (*e.g.*, C/C++, Java, PHP, JavaScript, python, etc.) do not have first-class support for space and time since they are designed to be general-purpose. The downside of the general-purpose approach is the complexity and size of code. Since these programming languages are imperative, meaning that we have to use *for, if*, or *while* to control the flow of the programs and explicitly handle state, the code can get large and complex easily.

In contrast to the general-purpose approach, if we know a specific problem domain very well and want to provide first-class support for key domain concepts (such as space and time), we can take a domain-specific approach. The

*Corresponding author
Email addresses: imais@cs.rpi.edu (Shigeru Imai), cvarela@cs.rpi.edu (Carlos A. Varela)

code written in the domain-specific approach is much simpler and more declarative as in logic programming languages [1] and therefore simpler to write, read, and reason about, but it is generally less expressive, than code written in general-purpose programming languages.

Operating an aircraft is a complicated task since there are a lot of complex correlations between the instruments in a cockpit. If some failure happens during a flight, it is not easy to find the cause of the failure by looking at the available (potentially partially erroneous) data , and also, a misinterpretation of instrument readings could even lead to a tragic accident [2]. We illustrate our programming model with a flight planning system that reports potential sources of data problems such as mechanical failures or extreme weather conditions. An explicit mathematical model of data co-relationships can with high probability signal data errors, potentially providing pilots with better information in emergency scenarios, allowing them to take appropriate actions in a timely manner, and ultimately reaching their destination safely and efficiently.

In this paper, we present our initial effort on designing a programming model for spatio-temporal data streaming applications, aiming to apply the model to a flight planning system. The model provides first-class support for space and time specific operations including data selection and interpolation when no data is available for a certain location and time.

2. Motivation

Auto-pilots cannot take the right action when the data they are receiving is out of date or incorrect. This may have led to the tragic crash of Air France flight 447, killing all 216 passengers and 12 aircrew [2]. The records from the crash have suggested that the pilots lost control of the airplane because they raised the nose of the airplane when it should not have been brought up. Many experts now understand that the airplane went into clouds with thunderstorms and its iced speed sensors provided inaccurate information to the autopilot, causing it to disengage. The pilots then incorrectly reacted to the emergency by raising the nose of the plane when in fact it needed to go down to break the stall.

An active redundant data-driven flight system may help prevent crashes caused by sensor or other data errors. For example, by comparing the airspeed data to the ground speed data, a flight system would be able to fact check a bad airspeed reading, assuming reasonable constraints on the wind speed. If the pilot is only operating by airspeed data alone, they would have no way of knowing that there is an error in the system and they would respond to the incorrect data, upsetting the balance of the plane. The ground speed data would instead provide a fact checking mechanism because if airspeed were swiftly changing, ground speed would be doing the same. If airspeed is changing, but ground speed remains unchanged, the more active flight system would be able to notify the pilot of the discrepancy, allowing for better informed decision making.

Our goal is to develop a data streaming programming model that makes explicit the connections between different spatio-temporal data streams. Flight systems developed using our model would make explicit the redundancies in the data and allow the different data streams to essentially fact check each other greatly reducing the possibility of accidents. The proposed spatio-temporal data streaming programming model can also be applied to other domains generating space and time specific data.

3. Spatio-Temporal Data Streaming Programming Model

3.1. Programming Model

Our proposed programming model is designed for applications that handle spatio-temporal input data streams as shown in Figure 1. In this model, the application gets data $(d'_1, d'_2, \ldots, d'_n)$ from the *data selection* module, which takes incoming data streams $(d_1, d_2, \ldots, d_n)$ as inputs, and then the application indefinitely generates outputs $(o_1, o_2, \ldots, o_m)$ and data errors $(e_1, e_2, \ldots, e_l)$ based on an *application model*. Each input data stream $d_i(x, y, z, t)$ is a function of location and time. The number of arguments of d_i varies depending on dimensions of the location information, that is, $d_i(t)$ for 0-D (*i.e.*, no location support), $d_i(x, t)$ for 1-D, $d_i(x, y, t)$ for 2-D, and $d_i(x, y, z, t)$ for 3-D. Some data streams are coming in real-time whereas some predicted information (*e.g.*, weather forecasts) is associated with future time periods.

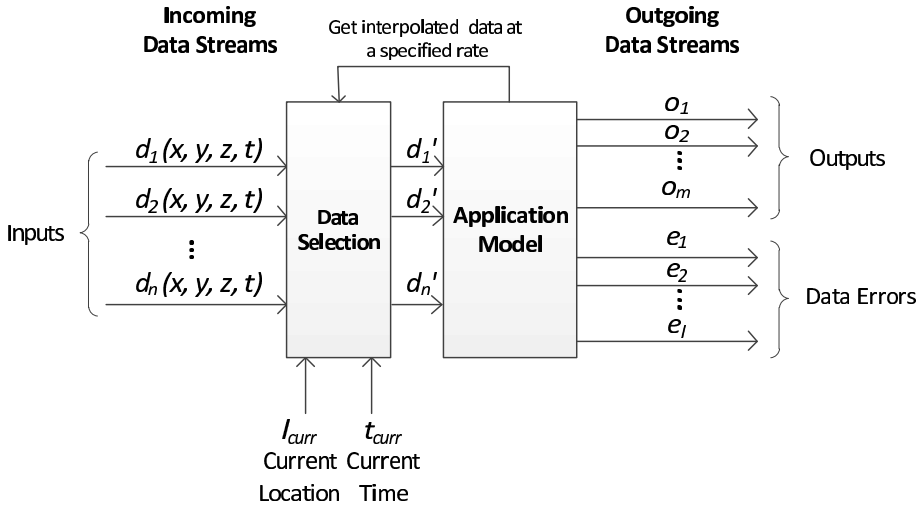


Figure 1: Programming model handling spatio-temporal input data streams

Table 1: An example of a weather forecast input data stream

Location $(lat_1, long_1) - (lat_2, long_2)$	Altitude	Time $(from) - (to)$	Chance of Icing
(42.73, -73.69)-(42.70, -73.66)	8000 ft.	01/30/2012:10:00-12:00 (GMT)	50%
(42.73, -73.69)-(42.70, -73.66)	8000 ft.	01/30/2012:12:00-14:00 (GMT)	60%
(42.73, -73.69)-(42.70, -73.66)	8000 ft.	01/30/2012:14:00-16:00 (GMT)	80%

Table 1 shows an example of a weather forecast input data stream coming to the Data Selection module. As noted from the table, the location information is given by regions where each region is represented by two horizontal locations (latitude, longitude) and an altitude, and the time information is given by time periods where each period is represented by two time points in GMT. Since not all the data is on the table, for example, chance of icing is not defined when the time is 01/30/2012:09:00 (GMT); however, we can define data by selecting or interpolating the existing data when no data is defined for a given location and time. In our programming model, the Data Selection module stores some amount of incoming data stream until it becomes out of date. The application acquires the selected or interpolated data $(d'_1, d'_2, \ldots, d'_n)$ from the Data Interpolation module at a certain rate specified in the application and computes both outputs and data errors. The application continues this computing process in an infinite loop until the user requests to stop the computation. The Data Selection module essentially allows an application to view a set of heterogeneous data streams as a homogeneous data stream, and therefore enables a separation of concerns: application programmers can focus on their application model.

3.2. Support for Spatio-Temporal Data Selection

We define two types of data selection and one data interpolation method for the location and time as shown below. These operations are applicable to either single variables (*i.e.* t, x, y, or z) or multiple variables (*i.e.* combinations of t, x, y, and z. By using these operations, application programmers can use locally related data even in the case when the given data is sparse.

- *closest*
  This method takes a 1-D argument (*i.e.,* t, x, y, or z) to find the data closest to a given location or time. Figure 2 shows examples of selecting closest data to the current time and location respectively. In Figure 2(a), when selecting the closest time to the current time t_{curr}, $d_i(t_{curr})$ is not defined, but $d_i(t)$ is defined for $\{t \mid t_1 \leq t \leq$

$t_2, t_3 \leq t \leq t_4, t_5 \leq t \leq t_6\}$. Since t_4 is closest to t_{curr}, we define $d'_i(t_{curr}) \triangleq d_i(t_4)$. Similarly, we define $d'_i(x_{curr}) \triangleq d_i(x_3)$ for the example shown in Figure 2(b).

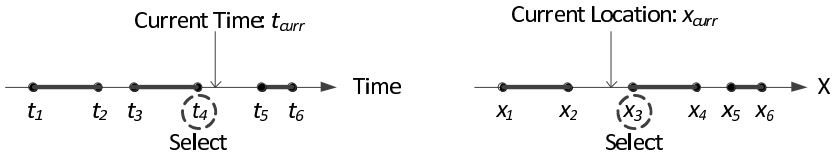


Figure 2: (a) Selecting the closest time; (b) Selecting the closest x value

- *euclidean*
  This method takes 2-D or 3-D arguments to find the data closest to a given location. Figure 3 shows an example for the 2-D case, where data is not defined for the current location $l_{curr} = (x_{curr}, y_{curr})$, but are defined for l_0, and l_1. Since l_{curr} is closest to $l_0 = (x_0, y_0)$ in Euclidean distance, we define $d'_i(x_{curr}, y_{curr}) \triangleq d_i(x_0, y_0)$.

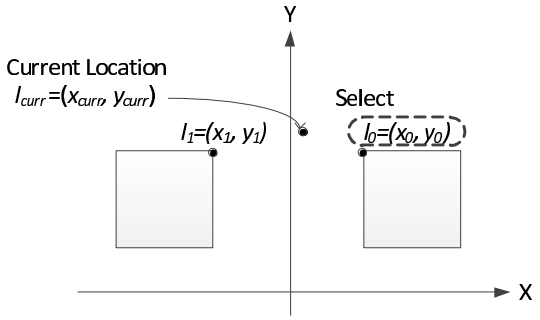


Figure 3: Selecting the closest 2D region in Euclidean distance

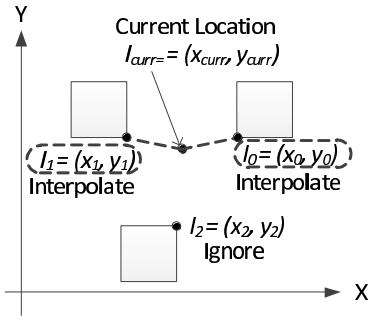


Figure 4: Linear interpolation

- *linear interpolation*
  This method takes 1-D, 2-D or 3-D arguments to interpolate the defined data. It also takes another argument n_{interp} to select closest n_{interp} data from a given location to interpolate. Suppose we have a situation shown in Figure 4, where data is not defined for the current location $l_{curr} = (x_{curr}, y_{curr})$, but are defined for l_0, l_1, and l_2. Also, suppose that $n_{interp} = 2$, we select l_0 and l_1 since they are closer to l_{curr} than l_2. In such a case, we linearly interpolate the data defined for l_0 and l_1 by taking a weighted sum based on the Euclidean distance as follows:

$$d'_i(x_{curr}, y_{curr}) \triangleq (1 - \frac{\|l_0 - l_{curr}\|}{\sum_{i=0}^{1} \|l_i - l_{curr}\|}) \cdot d_i(x_0, y_0) + (1 - \frac{\|l_1 - l_{curr}\|}{\sum_{i=0}^{1} \|l_i - l_{curr}\|}) \cdot d_i(x_1, y_1) \quad (1)$$

  Note that the equation (1) can be easily extended to n data points.

Multiple methods can be specified in the application program and they apply to the input data in order. If multiple data get selected by one method (*e.g.*, more than one closest point), a subsequent method takes that multiple data as the input and further select data. If there still remains more than one data after applying all the methods, then we implicitly apply linear interpolation to output the final value.

4. Spatio-Temporal Data Streaming Programming Language

In this section, we describe a spatio-temporal programming language by defining its grammar and showing two example programs.

4.1. Grammar Definition

A grammar definition for a declarative programming language following the proposed programming model is shown in Figure 5. A program (*Program*) consists of four parts: a program name, inputs, outputs, and errors. The program name is defined by a variable name (*Var*). The inputs can have multiple entries of *Input*, which is defined by one or more input variables (*Vars*), a dimension of the inputs (*Dim*), and a data selection method described in the previous section (*Method*). The outputs and errors are defined separately, but have the same output format (*Output*). Output is defined by one or more output variables (*Vars*), mathematical expressions (*Exps*), and a time interval to specify the frequency of the output (*Time*).

```
Program   ::=  program Var;
                      inputs Input*
                      outputs Output*
                      errors Output*
Input     ::=  Vars:  Dim using Methods;
Output    ::=  Vars:  Exps at every Time;
Dim       ::=  '(t)' | '(x,t)' | '(x,y,t)' | '(x,y,z,t)'
Methods   ::=  Method | Method, Methods
Method    ::=  (closest | euclidean | interpolate) '(' Exps ')'
Time      ::=  Number (nsec | usec | msec | sec | min | hour | day)
Exps      ::=  Exp | Exp, Exps
Exp       ::=  Func(Exps) | Exp Func Exp | '(' Exp ')' | Value
Func      =    { +, -, *, /, sqrt, sin, cos, tan, abs, ...}
Value     ::=  Number | Var
Number    ::=  Sign Digits | Sign Digits'.'Digits
Sign      ::=  '+' | '-' | ''
Digits    ::=  Digit | Digits Digit
Digit     =    { 0, 1, 2, ...,9 }
Vars      ::=  Var | Var, Vars
Var       =    { a, b, c, ...}
```

Figure 5: Spatio-temporal data streaming programming language grammar

4.2. Example Programs

4.2.1. Simple Example

Here we show one of the simplest programs implemented by the proposed programming model. It takes two input streams, $a(t)$ and $b(t)$, and outputs an error defined by $e = (b' - 2a')$ every 1 second as shown in Figure 6. Note that these two input streams are not associated to any location information.

An example specification of the above simple program is shown in Figure 7. In this example, there are two input data streams in which each stream is a function of time. The data selection method used in the program is specified by `closest(t)`, which means that the program instructs the Data selection module to select the closest t to the current time t_{curr}.

Errors behave differently depending on the input data streams, thus they tell us valuable information about the information sources. Figure 8 shows three different types of errors generated by simulations of the simple example program specification described in Figure 7. The assumption here is that every $1 \pm \epsilon$ seconds, the variable `a`'s input comes as $1\pm\epsilon, 2\pm\epsilon, 3\pm\epsilon, \ldots$ whereas the variable `b`'s input comes as $2\pm\epsilon, 4\pm\epsilon, 6\pm\epsilon, \ldots$, *i.e.*, they hold the mathematical relationship: `b = 2 * a`. In Figure 8(a), most of the time the error stays at zero, but there are several spikes due to transient fluctuation of the data input timing. It happens occasionally since the use of `closest(t)` causes the Data Selection module to select data at one second earlier or one second later than it is supposed to select. This type of

error is unavoidable without a special synchronization mechanism between multiple data streams. Figure 8(b) shows an example of the out-of-sync error. As shown in the graph, the error becomes consistently large at around 30 seconds of the simulation time. This is because the variable `a`'s input data stream becomes consistently one second behind the variable `b`'s input data stream. Figure 8(c) suggests more critical failure of the variable `a`'s input data source. At around 40 seconds of the simulation time, the error starts growing linearly. This linear increase of the error explains that the input data stream of the variable `a` stops coming after 40 seconds of the simulation time, which potentially means that a critical failure occurred at the source of the variable `a`.

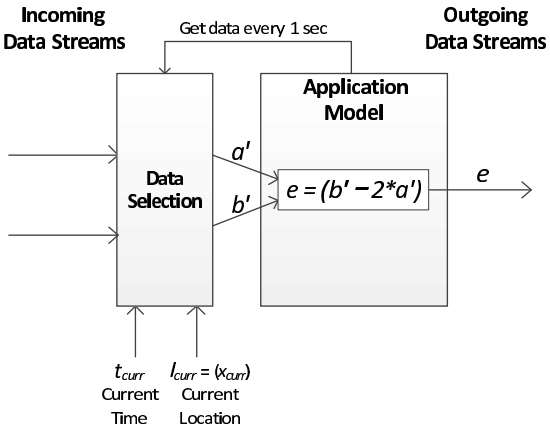


Figure 6: A simple application with temporal data streams

```
program twice;
    inputs
        a: (t) using closest(t);
        b: (t) using closest(t);
    outputs;
    errors
        e: (b - 2 * a)
            at every 1 sec;
end;
```

Figure 7: A simplest program specification

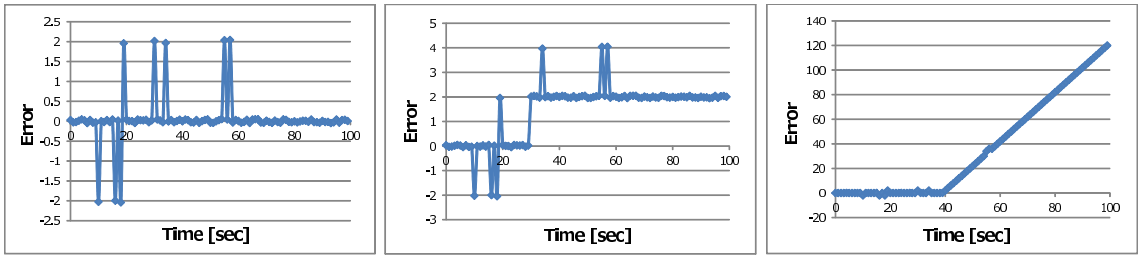


Figure 8: Examples of errors generated by a simple example: (a) Timing transient error; (b) Out-of-sync error; (c) Data input failure

4.2.2. *Flight Planning*

This is an example of a simplified flight planning system. Suppose that sensors in a plane record airspeeds v_a during a given flight and GPS units record the airplane's flight path over the ground including ground speeds v_g at different locations. An aircraft's airspeed and ground speed are related by the following mathematical formula: $v_g = \sqrt{v_a^2 + 2v_a \cdot v_w \cdot \cos(\alpha_a - \alpha_w) + v_w^2}$, where v_a and α_a are the aircraft airspeed and heading, and v_w and α_w are the wind speed and direction. Also, we can compute crosswind velocity: $v_x = v_w \cdot \sin(\alpha_a - \alpha_w)$. Therefore, given the aircraft desired course α_d, it is possible to compute the crab angle δ by using the formula (2) so that the aircraft can use $\alpha_a = \alpha_d + \delta$ as the heading to maintain the desired direction under varying wind conditions. The above mentioned relationship can be modeled as an application shown in Figure 9, which outputs the crab angle δ and error e that is the difference between the monitored ground speed v_g from GPS and the calculated one from v_a, v_w, α_a, and α_w.

$$\delta = \arcsin(v_x/v_g) = \arcsin\left(\frac{v_w \cdot \sin(\alpha_a - \alpha_w)}{\sqrt{v_a^2 + 2v_a \cdot v_w \cdot \cos(\alpha_a - \alpha_w) + v_w^2}}\right) \tag{2}$$

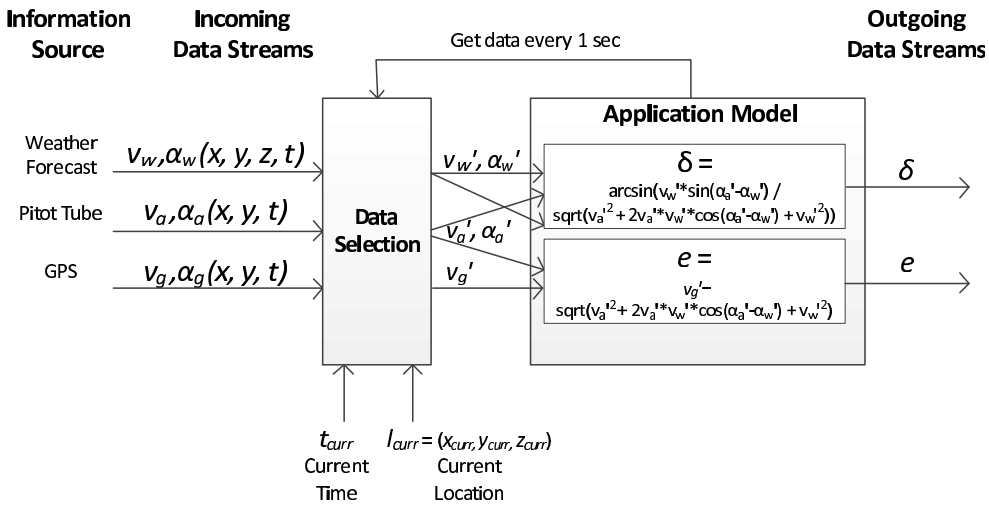


Figure 9: A flight planning application model using the spatio-temporal data streaming programming model

A code example of the flight planning application is shown in Figure 10. In this example, there are three input data streams in which each stream has two functions. Since each of these two functions has the same source of information and arguments, they are declared as a single input data stream. In the case of the first input data stream, there are two functions, `wind_speed(x,y,z,t)` and `wind_angle(x,y,z,t)`, that share the same arguments and information source (weather forecast). Data interpolation methods used for and `wind_speed(x,y,z,t)` and `wind_angle(x,y,z,t)` are specified by `euclidean(x,y)`, `closest(t)`, and `interpolate(z, 3)`. These methods apply in order: first, the closest x and y to x_{curr} and y_{curr} in Euclidean distance are selected; second, the closest t to the current time t_{curr} is selected; and finally, the final value is linearly interpolated on the z-axis using up to three closest data points to z_{curr} as specified in the argument.

```
program flightplan;
    inputs
        wind_speed, wind_angle: (x, y, z, t)
            using euclidean(x, y), closest(t), interpolate(z, 3);

        air_speed, air_angle: (x, y, z, t)
            using euclidean(x, y), closest(t);

        ground_speed, ground_angle: (x, y, z, t)
            using euclidean(x, y), closest(t);

    outputs
        crab_angle: arcsin(wind_speed * sin(wind_angle - air_angle) /
                           sqrt(air_speed^2 + 2 * air_speed * wind_speed *
                                cos(wind_angle - air_angle) + wind_speed^2))
            at every 1 sec;

    errors
        e: ground_speed - sqrt(air_speed^2 + 2 * air_speed * wind_speed *
                               cos(wind_angle - air_angle) + wind_speed^2))
            at every 1 sec;
end;
```

Figure 10: A declarative specification of the flight planning application

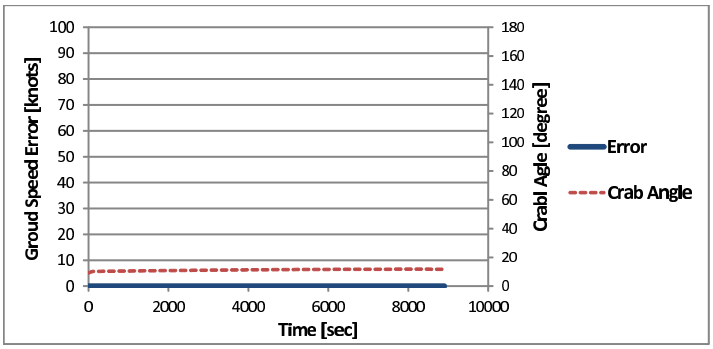


Figure 11: Normal conditions signature observed from a flight planning simulation

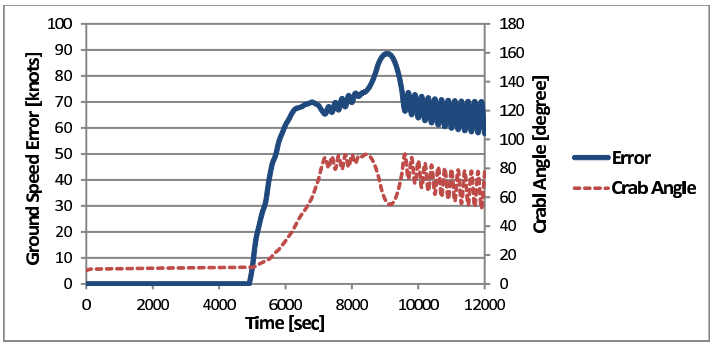


Figure 12: Pitot tube failure error signature observed from a flight planning simulation (the monitored airspeed starts decreasing at time = 5000 due to the pitot tube failure)

Example simulation results of the flight planning application generated from the program specification described in Figure 10 are shown in Figure 11 and Figure 12. In this simulation, a simple autopilot system navigates an airplane by using the crab angle received from the application. The airplane flies at 100 knots from Washington D.C. to Albany, which is 281 nautical miles (323 miles) away. The simulation also takes into account the effect of winds. In a normal condition, the autopilot successfully navigates the airplane to the destination with an ideal path as shown in Figure 11. The error of ground speed stays zero all through the simulation and the crab angle is almost constant (it actually slightly increases to adapt the wind speed changes). Figure 12 shows an error signature caused by a pitot tube failure. At time = 5000, a pitot tube starts icing and that causes the monitored airspeed to decrease gradually and gets almost zero eventually, while the airplane keeps flying at 100 knots. As seen from the graph, the autopilot's navigation does not work successfully this time. We can see a clear signature of the error here: the ground speed error grows quickly as soon as the airspeed start decreasing at time = 5000 and remains around 70 knots. The crab angle also grows up as the airspeed decreases.

As we can see from the simulation, a pilot can benefit from this application by following the crab angle to control the direction of the airplane, and also monitoring the error output to see if there is a mechanical failure of the instruments or the forecast information is wrong.

5. System Interaction

Figure 13 shows how a spatio-temporal application interacts with the system. The interactions are based on a client-server model using Internet sockets in which the application works as a server and takes inputs from the clients on a single port. It outputs some values to the specified output ports as well as error values to the specified error ports.

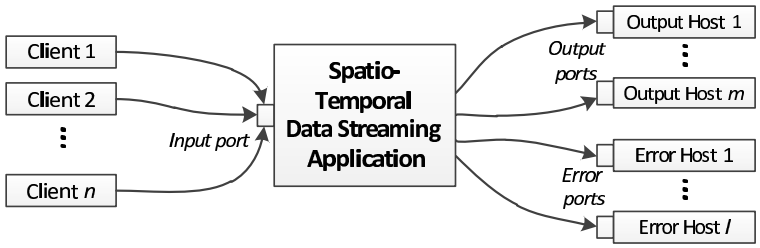


Figure 13: System interaction of a spatio-temporal data streaming application

When executing a binary object generated from the program specification, we specify input, output, and error ports as follows. In this example, the `flightplan` application illustrated in Section 4.2.2 takes an input data stream on the port `10001` and sends output and error streams to the hosts specified by `10.0.0.1:20001` and `10.0.0.2:30001` respectively.

```
$./flightplan -input 10001 -outputs 10.0.0.1:20001 -errors 10.0.0.2:30001
```

The input, output, and error data streams share the same format shown in Table 2. The first line is used to declare one or more variables (`var0, var1,...`) in a single data stream. The values of the declared variables start from the second line. The data stream can have multiple values (`value0, value1,...`) with various spatial and temporal combinations: ex1 is defined for a 3-D region and a time interval; ex2 is defined for a 2-D point and a time interval; ex3 is defined for a 1-D interval and a particular time; ex4 is defined for no location and a particular time. All lines have to end with an end-of-line marker (`\r\n`). Especially, the last line have to have only one end-of-line marker.

Note that the data format for input is compatible with output and error, that is, we can connect either an output or error port to an input port of another application.

Table 2: The data stream format of input, output, and error

first line	`#var0, var1,...\r\n`
after second line	ex1) `x0,y0,z0-x1,y1,z1:t0-t1:value0,value1,...\r\n`
	ex2) `x,y:t0-t1:value0,value1,...\r\n`
	ex3) `x0-x1:t:value0,value1,...\r\n`
	ex4) `:t:value0,value1,...\r\n`
last line	`\r\n`

6. Related Work

Spatio-temporal constraint logic programming has been proposed. STACLP [3] offers first-class support for representing and reasoning about spatial and temporal data. A similar logic language to STACLP, MuTACLP [4][5], is used to analyze geographical data especially for GIS (Geographical Information Systems). Both STACLP and MuTACLP are implemented based on a Prolog system. Programming languages that support probabilistic reasoning have also been proposed. PRISM[6] is a logic-based language that integrates logic programming and stochastic reasoning including parameter learning. PRISM is capable of parameter learning from a given set of data and estimates the probability to best explain the data. PRISM is also built on top of a Prolog system. Our proposed programming language is also highly declarative and it is to generate code that takes as inputs, spatio-temporal data streams.

There are programming languages for time-critical systems such as automatic control and monitoring systems. LUSTRE [7] [8], Giotto [9], and Esterel[10] are included in this category. These programming languages are declarative and designed to respond input events synchronously. Although their target systems are similar to ours, the main focus of these languages is real time behavior and there is no special support for spatial information nor reasoning capabilities.

7. Conclusions and Future Work

We present a programming model for spatio-temporal data streaming applications. In particular, it has first-class support for data selection and interpolation when no data is available for a given location and time. Towards our primary goal of applying this programming model to flight planning applications, we have several future research directions: 1) development of a compiler of the proposed language and a fully-functional application using real spatio-temporal data to demonstrate the advantage of the proposed programming model; 2) learning error signatures for common failures in aviation system, 3) adding the probability of data accuracy as inversely proportional to the spatio-temporal distance between the current location and time and the defined data points; 4) studying stochastic reasoning techniques and investigating the applicability of such techniques to spatio-temporal data streaming applications.

Acknowledgments

This research is partially supported by the Air Force Office of Scientific Research.

References

[1] J. W. Lloyd, Foundations of logic programming. Symbolic computation, Springer-Verlag, Berlin, Germany, 1984.
[2] Wikipedia, Air france flight 447, `http://en.wikipedia.org/wiki/Air_France_Flight_447`.
[3] A. Raffaet, T. W. Frhwirth, Spatio-temporal annotated constraint logic programming., in: PADL'01, 2001, pp. 259–273.
[4] P. Mancarella, G. Nerbini, A. Raffaet, F. Turini, MuTACLP: A language for declarative GIS analysis., in: Computational Logic'00, 2000, pp. 1002–1016.
[5] P. Baldan, P. Mancarella, A. Raffaet, F. Turini, MuTACLP: A language for temporal reasoning with multiple theories., in: Computational Logic: Logic Programming and Beyond'02, 2002, pp. 1–40.
[6] T. Sato, Y. Kameya, Parameter learning of logic programs for symbolic-statistical modeling, in: Journal of Artificial Intelligence Research(JAIR), Vol. 15, 2001, pp. 391–454.
[7] N. Halbwachs, P. Caspi, P. Raymond, D. Pilaud, The synchronous dataflow programming language LUSTRE, in: Proceedings of the IEEE, 1991, pp. 1305–1320.
[8] P. Caspi, D. Pilaud, N. Halbwachs, J. Plaice, Lustre: A declarative language for programming synchronous systems., in: POPL'87, 1987, pp. 178–188.
[9] T. A. Henzinger, B. Horowitz, C. M. Kirsch, Giotto: A time-triggered language for embedded programming., in: EMSOFT'01, 2001, pp. 166–184.
[10] G. Berry, The foundations of Esterel., in: Proof, Language, and Interaction'00, 2000, pp. 425–454.

Dynamic Data-Driven Avionics Systems with Stochastic Error Detection and Correction

Shigeru Imai and Carlos A. Varela

Abstract Dynamic Data-Driven Avionics Systems embody ideas from the Dynamic Data-Driven Applications Systems (DDDAS) paradigm by creating a data-driven feedback loop that continuously analyzes spatio-temporal data streams coming from airplane sensors, looks for errors in the data signaling different potential failure modes, and autonomously corrects for such erroneous data when possible. In this chapter, we define *error signatures* as constrained mathematical function patterns. These signatures help stochastically determine the true mode of operation of an avionics system. When a failure mode is detected, input data streams are corrected using redundant data in order to continue normal operation of the DDDAS avionics system. We introduce the *PILOTS* programming language to enable creation of DDDAS systems from high-level specifications of the relationships between data streams, error signatures, and error correction functions. We illustrate the applicability of PILOTS by showing how the Air France AF447 accident from June 2009 could have been prevented by using ground speed and wind speed to recompute air speed upon automatic detection of the pitot tubes icing failure. Aircraft accidents often happen as a result of a series of small problems chained in a sequence that compound themselves to become unmanageable. This work intends to attack one source of such accident chains: *data errors* that result from malfunctioning sensors, which can be automatically corrected thanks to the redundancy afforded by other instruments and physical and geometric models embedded in DDDAS systems.

1 Introduction

Operating airplanes is a difficult task; pilots have to keep making right decisions while dealing with a lot of information provided from the instruments in a cockpit.

Shigeru Imai and Carlos A. Varela
Department of Computer Science, Rensselaer Polytechnic Institute, 110 Eighth Street, Troy, NY 12180, USA, e-mail: {imais, cvarela}@cs.rpi.edu

Moreover, in the event of instrument failures, it becomes even more difficult because of potentially partially erroneous data. For example, pitot tubes icing which occurred to Air France flight 447 (AF447) in June 2009 led to faulty airspeed readings and eventually caused a fatal accident killing all 228 people on board [6]. The aircraft of the AF447 flight crashed in the Atlantic Ocean due to ice which temporarily formed in the pitot tubes causing erroneous airspeed readings, and the subsequent inability of the auto-pilot and human pilots to recover.

However, the faulty airspeed readings could have been prevented by endowing the avionics system with the ability to understand the following data relationship:

$$\overrightarrow{v_g} = \overrightarrow{v_a} + \overrightarrow{v_w}. \tag{1}$$

where $\overrightarrow{v_g}$, $\overrightarrow{v_a}$, and $\overrightarrow{v_w}$ represent the *ground speed*, the *airspeed*, and the *wind speed* vectors. These speeds are obtained through independent data collection methods: the ground speed is typically computed from Global Positioning System (GPS) data, the airspeed is computed from air pressure measurements by pitot tubes, and the wind speed from weather forecast computer models. Since any one of the three speeds can be calculated using the other two with Eq. (1), they are redundant to each other. If the auto-pilot was aware of such redundancy in the data, it could have fixed the incorrect airspeed readings quickly and thereby kept the system working properly. To facilitate the development of such smart avionics systems, we create the *Dynamic Data-Driven Avionics System* (see Fig. 1 for detail) based on the concept of *Dynamic Data-Driven Application Systems (DDDAS)* [8]. The Dynamic Data-Driven Avionics System is designed to dynamically correct erroneous data and interpolate sparse data.

PILOTS (**P**rogramm**I**ng **L**anguage for spati**O**-Temporal **S**treaming applications) [11, 15, 12] is a highly declarative programming language that embodies the concept of the Dynamic Data-Driven Avionics System. The PILOTS programming language enables high-level development of applications to handle spatio-temporal data streams and ultimately assist humans in making better decisions. Spatio-temporal data streams refer to data streams whose items include associated spatial and temporal coordinates, often viewed as meta data. Examples include temperature measurements, financial stock values, gas prices, surveillance camera imaging, and aircraft sensor readings. The PILOTS open-source project has evolved gradually to date [11, 15, 12]. In this chapter, we summarize the advancements of the project and its application to avionics systems based on PILOTS version 0.2.3 [17].

The rest of the chapter is organized as follows. Section 2 defines the requirements to realize the proposed Dynamic Data-Driven Avionics System and describes prior art of data streaming systems. Section 3 describes the methods for error detection and correction including the mathematical definition of error signatures. Section 4 shows the detailed design of the Dynamic Data-Driven Avionics System that we implemented as PILOTS. Section 5 shows an avionics application running on the PILOTS system and associated error signatures. Section 6 shows performance metrics and results of error detection performance for a private Cessna flight and the AF447 flight data. Finally, we conclude the chapter in Sect. 7 with future directions.

2 Research Challenges

2.1 Requirements for Dynamic Data-Driven Avionics System

As we have seen from the AF447 accident, spatio-temporal data streams may carry incorrect data from sensors. Furthermore, spatial and time density of data streams can be heterogeneous depending on the data sources. For example, GPS data may be produced every 100 ms whereas airspeed data may be produced every second. GPS data is associated with a single point due to its nature while weather forecast data is tied to a vast region in general. The Dynamic Data-Driven Avionics System shown in Fig. 1 is conceptually designed to deal with heterogeneous and potentially erroneous spatio-temporal data streams. Upon a request from the *Avionics Application*, the *Pre-Processor* takes raw data streams from sensors and then interpolates the streams into homogeneous and corrected ones. Thanks to the Pre-Processor, the Avionics Application can constantly compute its desired output with the corrected data. Since the Avionics Application controls how to process the raw data streams from the sensors, we can see that the resulting input data streams are dynamically steered by the Avionics Application. We would like to write applications in a domain-specific and declarative way so that application programmers can develop spatio-temporal data streaming applications reasonably easily.

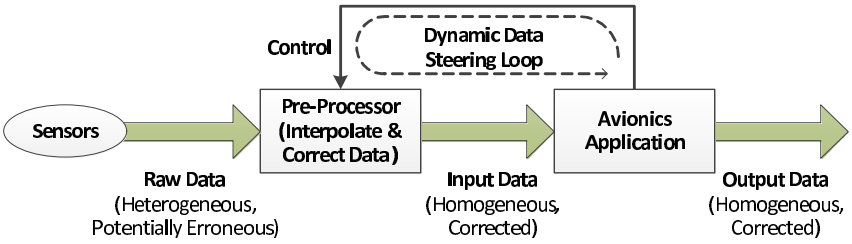


Fig. 1 Conceptual view of the Dynamic Data-Driven Avionics System

In summary, key requirements for the Dynamic Data-Driven Avionics System are described as follows:

1. Data interpolation and collection to view heterogeneous data streams as homogeneous data streams
2. Error detection and correction
3. High-level declarative programming language to write spatio-temporal applications including first-class support for describing how to control above two features

2.2 Prior Art

There are several systems that combine stream processing and data base management, *i.e.*, Data Stream Management Systems or DSMS, such as STREAM [5] and Aurora [1]. They are designed to execute SQL-like queries to unbounded continuous incoming data streams and output events of interest. Microsoft StreamInsight is a DSMS-based system and has been extended to support spatio-temporal streams [2]. Also, the concept of the moving object data base (MODB) which adds support for spatio-temporal data streaming to DSMS is discussed in [4]. These DSMS-based spatio-temporal stream management systems support general continuous queries for multiple moving objects; however, our streaming data analytics to detect errors based on signatures and correct data on the fly is beyond the scope of a purely declarative SQL-based query approach. In the context of Big Data processing, distributed, scalable, and fault-tolerant data streaming systems have been popular. Such systems include Storm [10], Spark Streaming [18], and S4 [13]. These systems are designed to be flexible and general so that complex applications such as machine learning or graph processing algorithms can run over a lot of distributed computers. Unlike their general approaches, our domain-specific approach enables highly declarative description of spatio-temporal data streaming applications. To the best of our knowledge, none of the existing streaming processing systems satisfies the requirements mentioned in Sec. 2.1.

3 Error Detection and Correction Methods

The error detection and correction methods are described in detail. The algorithm recognizes the shape of an error function using error signatures, identifies the type of error, and corrects an associated data value if possible.

3.1 Mathematical Preparations

Error function

An error function is an arbitrary function that computes a numerical value from independently measured input data. It is used to examine the validity of redundant data. If the value of an error function is zero, we interpret it as no error in the given data.

Figure 2 illustrates the relationship among the ground speed, airspeed, and wind speed when an airplane is flying. A vector $\overrightarrow{v}$ can be defined by a tuple (v, α), where v is the length of $\overrightarrow{v}$ and α is the angle between $\overrightarrow{v}$ and a base vector. Following this expression, $\overrightarrow{v_g}$, $\overrightarrow{v_a}$, and $\overrightarrow{v_w}$ are defined as (v_g, α_g), (v_a, α_a), and (v_w, α_w) respectively as shown in Fig. 2. To examine the relationship in Eq. (1), we can compute $\overrightarrow{v_g}$ by

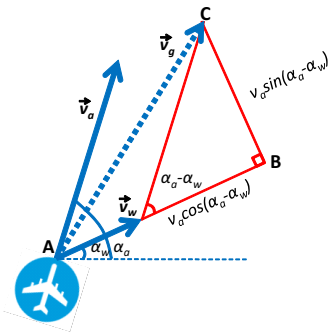


Fig. 2 Trigonometry applied to the ground speed, airspeed, and wind speed.

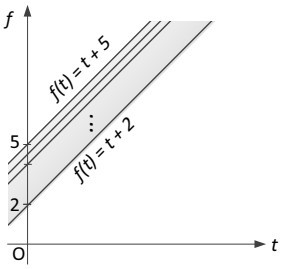


Fig. 3 Error signature S_I with a linear function $f(t) = t + k, 2 \leq k \leq 5$.

applying trigonometry to $\triangle ABC$. From measured v_g and computed v_g, we can define an error function as follows:

$$e(\overrightarrow{v_g}, \overrightarrow{v_a}, \overrightarrow{v_w}) = |\overrightarrow{v_g} - (\overrightarrow{v_a} + \overrightarrow{v_w})| = v_g - \sqrt{v_a^2 + 2v_a v_w \cos(\alpha_a - \alpha_w) + v_w^2}. \quad (2)$$

The values of input data are assumed to be sampled periodically from corresponding spatio-temporal data streams. Thus, an error function e changes its value as time proceeds and can also be represented as $e(t)$.

Error signatures

An *error signature* is a constrained mathematical function pattern that is used to capture the characteristics of an error function $e(t)$. Using a vector of constants $\bar{K} = \langle k_1, \ldots, k_m \rangle$, a function $f(t)$, and a set of constraint predicates $\bar{P} = \{p_1(\bar{K}), \ldots, p_l(\bar{K})\}$, the error signature $S(\bar{K}, f(t), \bar{P}(\bar{K}))$ is defined as follows:

$$S(\bar{K}, f(t), \bar{P}(\bar{K})) = \{f(t) | p_1(\bar{K}) \wedge \cdots \wedge p_l(\bar{K})\}. \quad (3)$$

For example, an interval error signature can be defined as:

$$S_I(\bar{K}, f(t), \bar{P}(\bar{K})) = \{f(t) = t + k \mid 2 \leq k \leq 5\}, \quad (4)$$

where $f(t) = t + k, \bar{K} = \langle k \rangle, \bar{P}(\bar{K}) = \langle 2 \leq k \leq 5 \rangle$. As shown in Fig. 3, this interval error signature S_I contains all linear functions with slope 1, and crossing the Y-axis at values $[2, 5]$

Given an error signature $S(\bar{K}, f(t), \bar{P}(\bar{K}))$, we enumerate its elements as *error signature samples*, *i.e.*,

$$s(t, \bar{K}) = f(t) \text{ s.t. } s(t, \bar{K}) \in S(\bar{K}, f(t), \bar{P}(\bar{K})). \quad (5)$$

An error signature sample is a particular function satisfying the constraints defined by an error signature. For the interval error signature S_I, a sample $s_I(t, \langle 3 \rangle)$ is $f(t) = t + 3$.

Mode likelihood vectors

Given a set of error signatures $\{S_0, \ldots, S_n\}$, we calculate $\delta_i(t)$, the distance between the measured error function $e(t)$ and each error signature S_i by:

$$\delta_i(t) = \min_{\bar{K}} \int_{t-\omega}^{t} |e(t) - s_i(t, \bar{K})| dt. \tag{6}$$

where ω is the window size and $s_i(t, \bar{K}) \in S_i$. Note that our convention is to capture "normal" conditions as signature S_0. The smaller the distance $\delta_i(t)$, the closer the raw data is to the theoretical signature S_i. We define the *mode likelihood vector* as $L(t) = \langle l_0(t), l_1(t), \ldots, l_n(t) \rangle$ where each $l_i(t)$ is defined as:

$$l_i(t) = \begin{cases} 1, & \text{if } \delta_i(t) = 0 \\ \frac{\min\{\delta_0(t), \ldots, \delta_n(t)\}}{\delta_i(t)}, & \text{otherwise.} \end{cases} \tag{7}$$

If $\delta_i(t) = 0$, it means that a measured error e and a error signature S_i is perfectly matched. Otherwise, we take the minimum of all the distances and divide it by $\delta_i(t)$ to normalize l_i. Consequently, each $l_i \in L, 0 < l_i \leq 1$ represents the ratio of the likelihood of signature S_i being matched with respect to the likelihood of the best signature.

3.2 Error Detection and Correction

Error Detection

Using a threshold $\tau \in (0, 1)$ and the previously defined likelihood vector L, an *error mode* is determined as follows:

- If there are more then one $l_i > \tau$, the error mode is *unknown*(-1),
- Otherwise, the error mode is i, where i is the index of the greatest element of L.

Because of the way L is created, the greatest element l_i will always be equal to 1. Given the threshold τ we check for one likely candidate that is sufficiently more likely than its successor by ensuring that $l_j \leq \tau$. Thus we determine the correct mode i by choosing the error signature S_i corresponding to l_i. If $i = 0$ then the system is in *normal mode.*

Error correction

It is problem dependent if a determined error mode i is recoverable or not. If there is a mathematical relationship between an erroneous value and other independently measured values, the erroneous value can be replaced by a new value computed from the other independently measured values. In the case of the speed example used in Eq.s (1) and (2), if the ground speed v_g is detected as erroneous, its corrected value v_g^c can be computed by the airspeed and wind speed as follows:

$$v_g^c = \sqrt{v_a^2 + 2v_a v_w \cos(\alpha_a - \alpha_w) + v_w^2}. \tag{8}$$

4 Dynamic Data-Driven Avionics Systems

PILOTS is a programming language and an associated runtime system specifically designed for analyzing data streams incorporating space and time. Using PILOTS, application developers can easily program an application that handles spatio-temporal data streams by writing a high-level (declarative) program specification. Also, by defining appropriate error signatures in the program specification, the PILOTS runtime system automatically detects and corrects errors in the data streams. This section describes the details of the PILOTS system as a concrete implementation of the Dynamic Data-Driven Avionics System.

4.1 System Overview

Figure 4 shows the architecture of the PILOTS runtime system, which implements the error detection and correction methods described in the previous section. It consists of three parts: the *Data Selection*, the *Error Analyzer*, and the *Application Model* modules.

The Application Model obtains homogeneous data streams $(d'_1, d'_2, \ldots, d'_N)$ from the Data Selection module, and then it generates outputs $(o_1, o_2, \ldots, o_M)$ and data errors $(e_1, e_2, \ldots, e_L)$. The Data Selection module takes heterogeneous incoming data streams $(d_1, d_2, \ldots, d_N)$ as inputs. Since this runtime is assumed to be working on a moving object, the Data Selection module is aware of the current location and time. Thus, it returns appropriate values to the Application Model by selecting or interpolating data in time and location depending on the data selection method specified in the PILOTS program.

The ErrorAnalyzer collects the latest ω error values from the Application Model and keeps analyzing errors based on a set of error signatures. If it detects a recoverable error, then it replaces an erroneous input with the corrected one by applying a corresponding error correction equation. The Application Model computes the outputs based on the corrected inputs produced from the Error Analyzer.

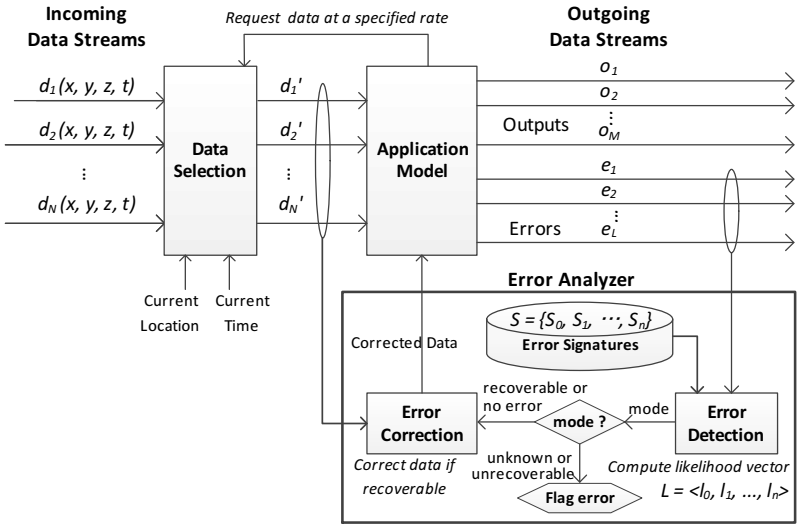


Fig. 4 Data streaming architecture with error detection and correction.

In the PILOTS programming model, the application acquires the selected or interpolated data $(d'_1, d'_2, \ldots, d'_n)$ from the Data Selection module at a certain rate specified in the application and computes both outputs and data errors. The application continues this computing process in an infinite loop until the user explicitly requests to stop the computation. The Data Selection module essentially allows an application to view a set of heterogeneous data streams as a homogeneous data stream, and therefore enables a separation of concerns: application programmers can focus on their application model.

4.2 Spatio-Temporal Data Selection

We define two types of data selection and one data interpolation method for the location and time. These operations are applicable to either single variables (*i.e.* t, x, y, or z) or multiple variables (*i.e.* combinations of t, x, y, and z).

- *closest*
  This method takes a 1-D argument (*i.e.,* t, x, y, or z) to find the data closest to a given location or time. Figure 5 shows examples of selecting closest data to the current time and location respectively. In Fig. 5(a), when selecting the closest time to the current time t_{curr}, $d_i(t_{curr})$ is not defined, but $d_i(t)$ is defined for $\{t \mid t_1 \leq t \leq t_2, t_3 \leq t \leq t_4, t_5 \leq t \leq t_6\}$. Since t_4 is closest to t_{curr}, we define $d'_i(t_{curr}) \triangleq d_i(t_4)$. Similarly, we define $d'_i(x_{curr}) \triangleq d_i(x_3)$ for the example shown in Fig. 5(b).

Fig. 5 **(a)** Selecting the closest time; **(b)** Selecting the closest x value

- *euclidean*
  This method takes 2-D or 3-D arguments to find the data closest to a given location. Figure 6 shows an example for the 2-D case, where data is not defined for the current location $l_{curr} = (x_{curr}, y_{curr})$, but are defined for l_0, and l_1. Since l_{curr} is closest to $l_0 = (x_0, y_0)$ in Euclidean distance, we define $d_i'(x_{curr}, y_{curr}) \triangleq d_i(x_0, y_0)$.

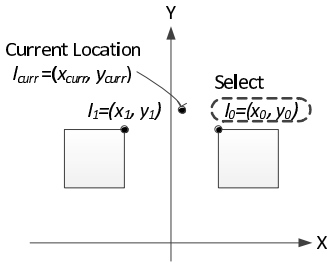


Fig. 6 Selecting the closest 2D region in Euclidean distance

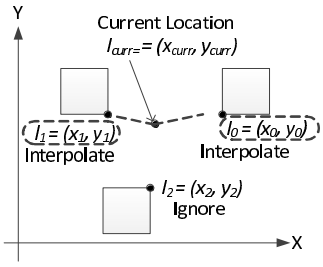


Fig. 7 Linear interpolation

- *linear interpolation*
  This method takes 1-D, 2-D or 3-D arguments to interpolate the defined data. It also takes another argument n_{interp} to select closest n_{interp} data from a given location to interpolate. Suppose we have a situation shown in Fig. 7, where data is not defined for the current location $l_{curr} = (x_{curr}, y_{curr})$, but are defined for l_0, l_1, and l_2. Also, suppose that $n_{interp} = 2$, we select l_0 and l_1 since they are closer to l_{curr} than l_2. In such a case, we linearly interpolate the data defined for l_0 and l_1 by taking a weighted sum based on the Euclidean distance as follows:

$$d_i'(x_{curr}, y_{curr}) \triangleq (1 - \frac{\|l_0 - l_{curr}\|}{\sum_{i=0}^{1} \|l_i - l_{curr}\|}) \cdot d_i(x_0, y_0) + (1 - \frac{\|l_1 - l_{curr}\|}{\sum_{i=0}^{1} \|l_i - l_{curr}\|}) \cdot d_i(x_1, y_1) \quad (9)$$

  Note that the equation (9) can be easily extended to n data points.

Multiple methods can be specified in the application program and they are applied to the input data in order. If multiple data get selected by one method (*e.g.*, more than one closest point), a subsequent method takes that multiple data as the input and further select data. If there still remains more than one data after applying all the methods, then we implicitly apply linear interpolation to output the final value.

4.3 Declarative Programming Language

A PILOTS program is highly declarative. It includes an `inputs` section to specify the data streams and how data is to be extrapolated from incomplete data, typically using declarative geometric criteria described in the previous subsection (*i.e.*., `closest`, `interpolate`, `euclidean` keywords) It also includes `outputs` and `errors` sections to specify the data streams to be produced by the application, as a function of the input streams with a given frequency. If a detected error is recoverable, output values are computed from corrected input data, otherwise original input data is used. The `signatures` and `correct` sections enable PILOTS programmers to specify error signatures for known error conditions, as well as the function to use to correct the data automatically if such data errors are found. An example PILOTS program is presented in Sect. 5.2.

5 Avionics System Application Example

In this section, we derive a set of error signatures for the speed example used in the previous sections. Also, we present a PILOTS program implementing the error signatures and corresponding error correction equations.

5.1 Error Signatures for Speed Data

We consider the following four operational modes: 1) normal (no error), 2) pitot tube failure due to icing, 3) GPS failure, 4) both pitot tube and GPS failures. Suppose an airplane is flying at airspeed v_a, we assume that other speeds as well as failed airspeed and ground speed can be expressed as follows.

- ground speed: $v_g \approx v_a$.
- wind speed: $v_w \leq a v_a$, where a is the maximum wind to airspeed ratio.
- pitot tube failed airspeed: $b_l v_a \leq v_a^f \leq b_h v_a$, where b_l and b_h are the lower and higher values of pitot tube clearance ratio and $0 \leq b_l \leq b_h \leq 1$. 0 represents a fully clogged pitot tube, while 1 represents a fully clear pitot tube.

- GPS failed ground speed: $v_g^f = 0$.

We assume that when a pitot tube icing occurs, it is gradually clogged and thus the airspeed data reported from the pitot tube also gradually drops and eventually remains at a constant speed while iced. This resulting constant speed is characterized by ratio b_l and b_h. On the other hand, when a GPS failure occurs, the ground speed suddenly drops to zero. This is why we model the failed ground speed as $v_g^f = 0$.

In the case of pitot tube failure, let the ground speed, wind speed, and airspeed be $v_g = v_a, v_w = av_a$, and $v_a^f = bv_a$. The error function (2) can be expressed as follows:

$$e = v_a - \sqrt{v_a^2(b^2 + 2ab\cos(\alpha_a - \alpha_w) + a^2)}.$$

Since $-1 \leq \cos(\alpha_a - \alpha_w) \leq 1$, the error is bounded by the following:

$$\begin{aligned} v_a - \sqrt{v_a^2(a+b)^2} \leq e \leq v_a - \sqrt{v_a^2(a-b)^2} \\ (1-a-b)v_a \leq e \leq (1-|a-b|)v_a. \end{aligned} \tag{10}$$

In the case of GPS failure, let the ground speed, wind speed, and airspeed be $v_g^f = 0, v_w = av_a$, and $v_a = v_a$. The error function (2) can be expressed as follows:

$$e = 0 - \sqrt{v_a^2(1 + 2a\cos(\alpha_a - \alpha_w) + a^2)}.$$

Similarly to the pitot tube failure, we can derive the following error bounds:

$$-(a+1)v_a \leq e \leq -|a-1|v_a. \tag{11}$$

We can derive error bounds for the normal and both failure cases similarly. Applying the wind to airspeed ratio a and the pitot tube clearance ratio $b_l \leq b \leq b_h$ to the constraints obtained in Inequations (10) and (11), we get the error signatures for each error mode as shown in Table 1.

Table 1 Error signatures for speed data.

Mode	Error Signature	
	Function	Constraints
Normal	$e = k$	$k \in [-av_a, av_a]$
Pitot tube failure	$e = k$	$k \in [(1-a-b_h)v_a, (1-\|a-b_l\|)v_a]$
GPS failure	$e = k$	$k \in [-(a+1)v_a, -\|a-1\|v_a]$
Both failures	$e = k$	$k \in [-(a+b_h)v_a, -\|a-b_l\|v_a]$

5.2 Speed Checker Program

A PILOTS program called `speedcheck` implementing the error signatures shown in Table 1 is presented in Fig. 8. This program checks if the wind speed, airspeed, and ground speed are correct or not, and computes a crab angle, which is used to adjust the direction of the aircraft to keep a desired ground track. The speed parameters used in this particular example are $a = 0.1, b_l = 0.2$, and $b_h = 0.33$, which are reasonable values from actual failure data we have observed. Also, for this program to be applicable to a Cessna 182-RG, we use a cruise speed of 162 knots as v_a.

6 Evaluation

We apply the error signatures defined in Sect. 5.1 to two sets of real flight data. The first one is a private flight using a Cessna 182-RG identified by N756VH [9] from Albany, NY to Fort Meade, MD on April 3rd, 2012. The other is the Air France flight 447 using an Airbus A330-203 which took off from Rio de Janeiro bound for Paris on June 1st, 2009. To simulate the failures mentioned in Sect. 5.1, we added corresponding errors to the N756VH Cessna flight data; however, we used the real pitot tube failure data for the AF447 flight. PILOTS programs' error detection accuracy and response time to mode changes are evaluated.

6.1 Performance Metrics

- **Accuracy**: This metric is used to evaluate how accurately the algorithm determines the true mode. Assuming the true mode transition $m(t)$ is known for $t = 0,1,2,\ldots,T$, let $m'(t)$ for $t = 0,1,2,\ldots,T$ be the mode determined by the error detection algorithm. We define $accuracy(m,m') = \frac{1}{T}\sum_{t=0}^{T} p(t)$, where $p(t) = 1$ if $m(t) = m'(t)$ and $p(t) = 0$ otherwise.
- **Average Response Time**: This metric is used to evaluate how quickly the algorithm reacts to mode changes. Let a tuple (t_i, m_i) represent a mode change point, where the mode changes to m_i at time t_i. Let $M = \{(t_1,m_1),(t_2,m_2),\ldots,(t_N,m_N)\}$ and $M' = \{(t'_1,m'_1),(t'_2,m'_2),\ldots,(t'_{N'},m'_{N'})\}$ be the sets of true mode changes and detected mode changes respectively. For each $i = 1\ldots N$, we can find the smallest t'_j such that $(t_i \leq t'_j) \wedge (m_i = m'_j)$; if not found, let t'_j be t_{i+1}. The response time r_i for the true mode m_i is given by $t'_j - t_i$. We define the average response time by $\frac{1}{N}\sum_{i=1}^{N} r_i$.

```
program speedcheck;
    inputs
    wind_speed, wind_angle (x,y,z,t) using
        euclidean(x,y), closest(t), interpolate(z,2);
    air_speed, air_angle (x,y,t) using
        euclidean(x,y), closest(t);
    ground_speed, ground_angle (x,y,t) using
        euclidean(x,y), closest(t);

    outputs
    crab_angle: arcsin(wind_speed *
                       sin(wind_angle - air_angle) /
                       sqrt(air_speed^2 + 2 * air_speed *
                            wind_speed *
                            cos(wind_angle - air_angle) +
                            wind_speed^2))
        at every 1 sec;

    errors
    e: ground_speed -
       sqrt(air_speed^2 + wind_speed^2 + 2 * air_speed *
            wind_speed * cos(wind_angle - air_angle));

    signatures
    /* v_a = 162 knots */
    S0(k): e=k,  -16.2<=k, k<=  16.2 "Normal";
    S1(k): e=k,   91.8<=k, k<= 145.8 "Pitot tube failure";
    S2(k): e=k, -178.2<=k, k<=-145.8 "GPS failure";
    S3(k): e=k,  -70.2<=k, k<= -16.2 "Both failures";

    correct
    S1: air_speed = sqrt(ground_speed^2 + wind_speed^2 +
                         2 * ground_speed * wind_speed *
                         cos(ground_angle - wind_angle));
    S2: ground_speed = sqrt(air_speed^2 + wind_speed^2 +
                            2 * air_speed * wind_speed *
                            cos(wind_angle - air_angle));
end
```

Fig. 8 A declarative specification of the `speedcheck` PILOTS program.

6.2 Experiment 1: N756VH Cessna Flight

6.2.1 Flight data

Flight data is collected through the following independent sources:

- **ground speed**: Flight track log provided by FlightAware [9].
- **airspeed**: Manually recorded by the pilot.
- **wind speed**: Weather forecast information provided by National Weather Service [14].

The flight duration is 1 hour 41 minutes. The collected speed data and error computed by Eq. (2) are shown in Fig. 9(a). Notice that the airspeed data during take off and landing is not accurate due to the data collection mechanism.

6.2.2 Experimental Settings

Using the `speedcheck` PILOTS program shown in Fig. 8, the 6060 seconds (=1 hour 41 minutes) of flight departing from Albany, NY and landing at Fort Meade, MD are recreated. Three types of error are simulated as shown below. In each case, all data streams except for erroneous one(s) are actual. Defined error modes are: -1 for unknown, 0 for normal, 1 for pitot tube failure, 2 for GPS failure, and 3 for both failures.

- **Pitot tube failure**: 2400 seconds after the departure, the airspeed drops from 162 knots to 50 knots within 10 seconds and stays at 50 knots until landing. The set of true mode changes is given by $M = \{(1,0),(2401,1)\}$.
- **GPS failure**: 2400 seconds after the departure, the ground speed drops from 171 knots to 0 knots immediately and stays at 0 knots until landing. The set of true mode changes is given by $M = \{(1,0),(2401,2)\}$.
- **Both pitot tube and GPS failures**: The above two speed changes happen simultaneously at 2400 seconds after the departure. Both speeds remain failed until landing. The set of true mode changes is given by $M = \{(1,0),(2401,3)\}$.

6.2.3 Results

For all the three cases, when $\omega = 1$ and $\tau = 0.8$, the best results are observed as follows: accuracy = 0.9294 and response time = 4 seconds for the pitot tube failure, accuracy = 0.935 and response time = 0 seconds for the GPS failure, and accuracy = 0.9342 and response time = 5 seconds for both failures. The transitions of the corrected speed and detected modes when $\omega = 1$ and $\tau = 0.8$ are shown in Fig.s 9(b) for the pitot tube failure, 9(c) for the GPS failure, and 9(d) for the both failures respectively. For the first 390 seconds, the error mode is detected incorrectly in all three cases; the true modes are 0 (normal mode) whereas the detected modes are 3 (both failures) during this period. These wrong mode detections are originated from the erroneously recorded airspeed. Other than that, the error detection method works pretty well for all three cases.

Detected modes go into the unknown mode for a short period around 2401 seconds for both pitot tube failure and both failures. Since the airspeed takes a few seconds to drop, during that time, the normal and pitot tube failure modes are competing against each other for the pitot tube failure case. For the both failures case, the GPS failure and both failures modes are competing. Unlike the other two cases, the ground speed drops immediately for the GPS failure, and there is no conflict with other error modes, thus the GPS failure mode is correctly detected without going into the unknown mode.

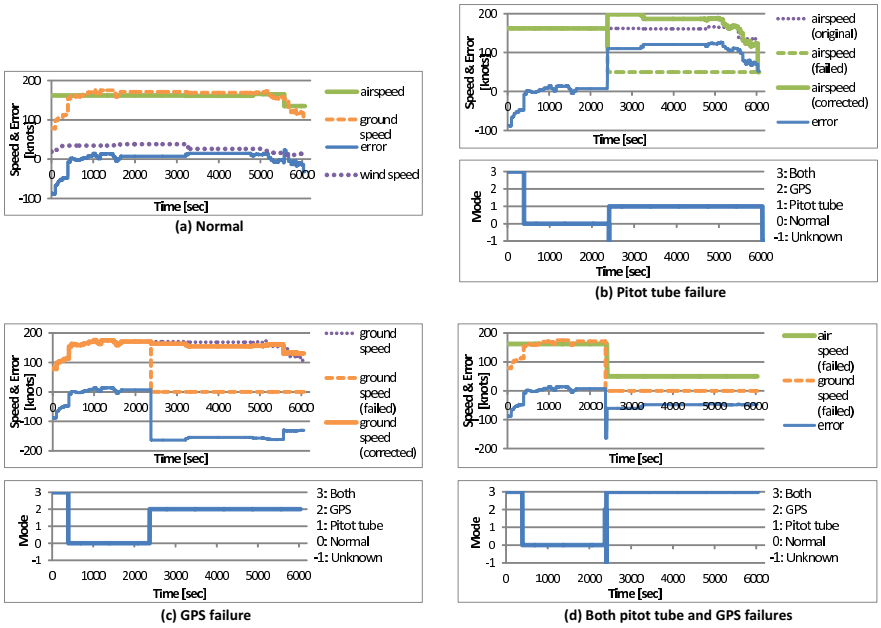


Fig. 9 Corrected speeds and detected modes for the N756VH 03-Apr-2012 KALB-KFME flight ($\tau = 0.8, \omega = 1$).

6.3 Experiment 2: Air France Flight 447

6.3.1 Flight Data

The ground speed and airspeed are collected based on Appendix 3 in the final report of Air France flight 447 [6]. Note that the (true) airspeed was not recorded in the flight data recorder so that we computed it from recorded Mach (M) and static air temperature (SAT) data. The airspeed was obtained by using the relationship: $v_a = a_0 M \sqrt{SAT/T_0}$, where a_0 is the speed of sound at standard sea level (661.47 knots) and T_0 is the temperature at standard sea level (288.15 Kelvin). Independent wind speed information was not recorded either. According to the description from page 47 of the final report: "(From the weather forecast) the wind and temperature charts show that the average effective wind along the route can be estimated at approximately *ten knots tail-wind*". We followed this description and created the wind speed data stream as ten knots tail wind.

6.3.2 Experimental Settings

According to the final report, speed data was provided from 2:09:00 UTC on June 1st 2009 and it became invalid after 2:11:42 UTC on the same day. Thus, we examine the valid 162 seconds of speed data including a period of pitot tube failure which occurred from 2:10:03 to 2:10:36 UTC. We also use the `speedcheck` PILOTS program shown in Fig. 8 except for constraints values in *signatures* which use $v_a = 470$ knots, the cruise airspeed of the AF447 flight. Defined error modes are the same as Experiment 1, so the set of true mode changes is defined as $M = \{(1,0),(64,1),(98,0)\}$.

6.3.3 Results

Same as Experiment 1, the best results, accuracy = 0.9631 and average response time = 2.5 seconds, are observed when $\omega = 1$ and $\tau = 0.8$. The transitions of the corrected speed and detected modes that show the best accuracy with $\omega = 1$ and $\tau = 0.8$ are shown in Fig. 10. Looking at the detected modes in Fig. 10, the pitot tube failure is successfully detected from 69 to 97 seconds except for the interval 64 to 69 seconds due to the slowly decreasing airspeed. The response time for the normal to pitot tube failure mode is 5 seconds and for the pitot tube failure to normal mode is 0 seconds (thus the average response time is 2.5 seconds). From the corrected airspeed in Fig. 10, the airspeed successfully starts to get corrected at 69 seconds and seamlessly transitions back to the normal airspeed when it recovers at 98 seconds.

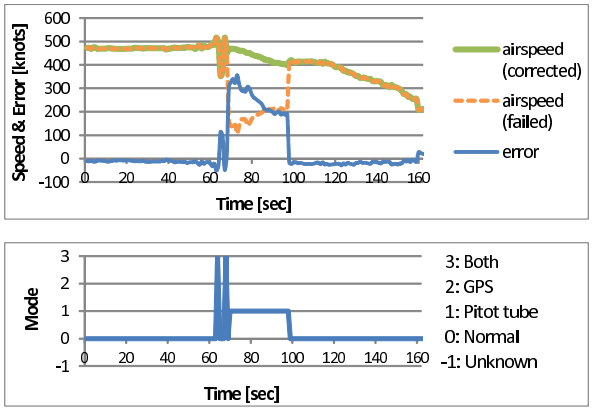


Fig. 10 Corrected airspeed and detected modes for AF447 flight.

7 Conclusion and Future Directions

We present the concept of the Dynamic Data-Driven Avionics System and its realization, the PILOTS system. The PILOTS runtime system dynamically interpolates and corrects heterogeneous data, and then provides it as homogeneous data to applications. This process can be viewed as dynamic incorporation of (interpolated and corrected) sensor data into an application. Also, since the application can control this process by specifying data interpolation methods and error signatures, we see that the application dynamically steers the data pre-processing process (*i.e.*, interpolation and correction). Thus, there is a dynamic feedback & control loop between the data pre-processing and the application, which is the core concept of Dynamic Data Driven Applications Systems (DDDAS).

The results for both Cessna and AF447 flight experiments illustrate the effectiveness of our approach. Since the system dynamically adapts to sparse and partially incorrect data, the application keeps generating valid outputs with a relatively simple PILOTS program as shown in Fig. 8. The error detection accuracy is at least 93% and the response time to correct data is at most 5 seconds.

When computing mode likelihood vectors, time to compute distances by Eq. (6) can be significant due to the exponential growth of the search space as the size of the constants set $\bar{K}$ increases. To use the presented error detection and correction methods in larger-scale real-time systems, techniques to bound the running time must be devised. Other future research directions include applying the error signature-based error correction methods to other flight accidents, *e.g.*, those due to fuel sensor reading errors. Also, uncertainty quantification [3] is an important future direction to associate confidence to data and error estimations in support of decision making. More and more data are expected to be available in cockpits in the near future [16], and thus automated data analysis systems will become even more crucial to both manned and unmanned aerial vehicles. We envision scalable smarter avionics systems processing massive data in real-time by dynamically creating and connecting multiple PILOTS program instances. Such systems need to reason about spatial and temporal data and constraints and give the pilots better information to make more accurate judgments during critical moments. The presented techniques and software can be used as a promising starting point to develop these dynamic data-driven avionics systems.

Acknowledgements This research is partially supported by the DDDAS program of the Air Force Office of Scientific Research, Grant No. FA9550-11-1-0332 and a Yamada Corporation Fellowship.

References

1. Abadi, D.J., Carney, D., Çetintemel, U., Cherniack, M., Convey, C., Lee, S., Stonebraker, M., Tatbul, N., Zdonik, S.: Aurora: a new model and architecture for data stream management. The

VLDB JournalThe International Journal on Very Large Data Bases **12**(2), 120–139 (2003)
2. Ali, M.H., Chandramouli, B., Raman, B.S., Katibah, E.: Spatio-temporal stream processing in Microsoft StreamInsight. IEEE Data Eng. Bull. pp. 69–74 (2010)
3. Allaire, D., Willcox, K.: Surrogate modeling for uncertainty assessment with application to aviation environmental system models. AIAA journal **48**(8), 1791–1803 (2010)
4. An, K., Kim, J.: Moving objects management system supporting location data stream. In: Proceedings of the 4th WSEAS International Conference on Computational Intelligence, Man-Machine Systems and Cybernetics, CIMMACS'05, pp. 99–104. World Scientific and Engineering Academy and Society (WSEAS), Stevens Point, Wisconsin, USA (2005)
5. Arasu, A., Babcock, B., Babu, S., Cieslewicz, J., Ito, K., Motwani, R., Srivastava, U., Widom, J.: Stream: The Stanford data stream management system. In: ACM SIGMOD Conference. Springer (2004)
6. Bureau d'Enquêtes et d'Analyses pour la Sécurité de l'Aviation Civile: Final Report: On the accident on 1st June 2009 to the Airbus A330-203 registered F-GZCP operated by Air France flight AF 447 Rio de Janeiro - Paris. http://www.bea.aero/en/enquetes/flight.af.447/rapport.final.en.php (2012)
7. Chandrasekaran, S., Cooper, O., Deshpande, A., Franklin, M.J., Hellerstein, J.M., Hong, W., Krishnamurthy, S., Madden, S.R., Reiss, F., Shah, M.A.: TelegraphCQ: continuous dataflow processing. In: Proceedings of the 2003 ACM SIGMOD international conference on Management of data, pp. 668–668. ACM (2003)
8. Darema, F.: Dynamic data driven applications systems: A new paradigm for application simulations and measurements. In: Computational Science-ICCS 2004, pp. 662–669. Springer (2004)
9. FlightAware: Flight track log for N756VH on 03-Apr-2012 (KALB-KFME). http://flightaware.com/live/flight/N756VH/history/20120403/1800Z/KALB/KFME/tracklog
10. software foundation, T.A.: Storm, distributed and fault-tolerant realtime computation. http://storm.incubator.apache.org/
11. Imai, S., Varela, C.A.: A programming model for spatio-temporal data streaming applications. In: Dynamic Data-Driven Application Systems (DDDAS 2012), pp. 1139–1148. Omaha, Nebraska (2012)
12. Klockowski, R.S., Imai, S., Rice, C., Varela, C.A.: Autonomous data error detection and recovery in streaming applications. In: Dynamic Data-Driven Application Systems (DDDAS 2013) Workshop, pp. 2036–2045 (2013)
13. Neumeyer, L., Robbins, B., Nair, A., Kesari, A.: S4: Distributed stream computing platform. In: Data Mining Workshops (ICDMW), 2010 IEEE International Conference on, pp. 170–177. IEEE (2010)
14. NOAA's National Weather Service: Forecast winds and temps aloft. http://aviationweather.gov/products/nws/winds/
15. S. Imai and C. A. Varela: Programming spatio-temporal data streaming applications with high-level specifications. In: 3rd ACM SIGSPATIAL International Workshop on Querying and Mining Uncertain Spatio-Temporal Data (QUeST) 2012. Redondo Beach, California, USA (2012)
16. U.S. Department of Transportation Federal Aviation Administration: Code of federal regulations part 91.225: Automatic dependent surveillance-broadcast (ADS-B) out performance requirements to support air traffic control (ATC) service; final rule. http://www.faa.gov/regulations_policies/faa_regulations/ (2013)
17. Worldwide Computing Laboratory, Rensselaer Polytechnic Institute: The PILOTS programming language. http://wcl.cs.rpi.edu/pilots/
18. Zaharia, M., Das, T., Li, H., Shenker, S., Stoica, I.: Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters. In: Proceedings of the 4th USENIX conference on Hot Topics in Cloud Ccomputing, pp. 10–10. USENIX Association (2012)